2-1-1994

VHDL modeling and design of an asynchronous version of the MIPS R30000 microprocessor

Paul Fanelli

VHDL MODELING AND DESIGN OF AN ASYNCHRONOUS VERSION OF THE MIPS R3000 MICROPROCESSOR

by

Paul Fanelli

A Thesis Submitted
in
Partial Fulfillment of the
Requirements for the Degree of
MASTER OF SCIENCE
in
Computer Engineering

Approved by:

Graduate Advisor - Prof. George A. Brown

Department Chairman - Dr. Roy Czernikowski

Reader - Dr. Tony Chang

DEPARTMENT OF COMPUTER ENGINEERING
COLLEGE OF ENGINEERING
ROCHESTER INSTITUTE OF TECHNOLOGY
ROCHESTER, NEW YORK

FEBRUARY, 1994

THESIS RELEASE PERMISSION FORM

ROCHESTER INSTITUTE OF TECHNOLOGY
COLLEGE OF ENGINEERING

Title: VHDL Modeling and Design of an Asynchronous Version of the MIPS R3000 Microprocessor.

I, Paul Fanelli, hereby deny permission to the Wallace Memorial Library of RIT to reproduce my thesis in whole or in part.

Date: 2/8/94

ABSTRACT

The goal of this thesis is to demonstrate the feasibility of converting a synchronous general purpose microprocessor design into one using an asynchronous methodology. This thesis is one of three parts that details the entire design of an asynchronous version of the MIPS R3000 microprocessor. The design includes the main architectural features of the R3000: the 5-stage pipeline, the thirty-two 32-bit register bank, and the 32-bit address and data paths. To limit the size of the project, the memory and coprocessor are excluded. Therefore, this design has implemented the entire set of instructions from the original synchronous version with the exception of the coprocessor support instructions.

The three participants in this project are Paul Fanelli, Kevin Johnson, and Scott Siers. Paul Fanelli developed the Very High Speed Integrated Circuit Hardware Description Language (VHDL) models for the processor. Three models, behavioral, dataflow, and structural, were constructed. Kevin Johnson designed the register bank, the arithmetic logic unit, and the shifter, including schematic diagrams and layouts. Scott Siers designed the pipeline stages, the multiplier/divider, the exception handler, and the completion signal generator, including schematic diagrams and layout. Each of the participants has written a separate thesis that covers one part of the total design.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

GLOSSARY OF TERMS

AA	Address Adder, hardware unit that calculates the address of next instruction, located in the instruction decode stage of pipeline
Accusim	Mentor Graphics Corporation analog circuit simulator
ADD8	Unit inside the ALU stage that calculates the link address for branch conditional instructions
ALU	Arithmetic Logic Unit, third stage of the pipeline
ALUB	Arithmetic Logic Unit Block, one of the units that comprises the ALU stage; consists of the ALUC, ALUC decoder, shifter, shifter control, branch control, etc.
ALUC	Arithmetic Logic Unit Component, unit that performs addition, subtraction, and logical computations
bcond	Branch Conditional group of instructions
BCB	Bus Control Block, unit inside the ALUB that controls what gets placed on the A and B busses in the ALU stage
BCU	Bus Control Unit, unit that controls which stage (IF or MEM) is granted access to the address and data busses
BJBOX	Branch and Jump Box, unit inside the ID stage that controls what gets sent to the AA
bton	Bits-to-Natural function, converts bits to natural numbers
CMOS	Complementary Metal Oxide Semiconductor
CPU	Central Processing Unit
current_inst	Variable that holds the current instruction
DBOX	Dirty Box, unit inside the ID stage that handles data dependencies
DCVSL	Differential Cascade Voltage Switch Logic

DOD	Department of Defense
ea	Variable that holds the effective address, used to calculate effective address of load and store instructions
EH	Exception Handler, unit that handles exceptions from the IF, ID, ALU, and MEM stages and generates an interrupt vector that is sent to the IF
FLOW	Reorders a program that contains branch and jump instructions to follow the program flow.
FSM	Finite State Machine
funct	Function field that holds the minor operation code used for instruction decoding
GPR	General Purpose Register
HCC	Handshaking Control Circuit
ibo	Immediate or base-offset, one of the output signals of the instruction decoder in the ID stage that goes high either for an immediate instruction or for an instruction that needs a base-offset calculation
ID	Instruction Decode, second stage of pipeline
IE	Instruction Execution
IF	Instruction Fetch, first stage of pipeline
immed	Field that holds the immediate value used for instruction decoding
LSB	Least Significant Bit
MASS	MIPS Assembler, program written to convert MIPS assembly code into machine code
MDU	Multiplier/Divider Unit
MEM	Memory, fourth stage of pipeline
MERA	MIPS Expected Results Assembler, program written to help generate an expected results file used in conjunction with each models test bench

MIPS	Name of company that designed, developed, and built the R3000 microprocessor, also stands for Millions of Instructions Per Second which is a processor speed rating
MPP	MIPS Preprocessor, program written to load processor with test programs
MSB	Most Significant Bit
MU	Mask Unit, unit inside the MEM stage that determines the length of the requested data and whether the data is sign or zero-extended
ns	Nanoseconds
offset	Field that holds the offset value used for instruction decoding
op	Field that holds the major operation code used for instruction decoding
opcode	Operation Code
PC	Program Counter
pc_reg	Variable that holds the program counter value
R3000	Model number of processor this thesis is modeling
rd	Destination register field used for instruction decoding
RISC	Reduced Instruction Set Computer
rs	Source register field used for instruction decoding
rt	Target register field used for instruction decoding
shamt	Field that holds the shift amount
special	Special Instructions, name of group of MIPS instructions
SU	Shift Unit, unit inside the MEM stage that shifts data when it is not aligned on a word boundary
target	Field that holds the target value used for instruction decoding

TRDS	Target Register Dirty Select, unit that selects which register will be set dirty
vbt	Valid Byte Tag, this signal is generated by the decoder inside the MEM stage and is used by the GPR bank in the ALU stage; it specifies which bytes of data are valid and can be written back to a register
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VLSI	Very Large Scale Integration
WB	Writeback, fifth stage of pipeline

1.0 INTRODUCTION

This thesis is one of three parts encompassing the modeling, design, and implementation of an asynchronous microprocessor. This project was performed cooperatively by Scott Siers, Kevin Johnson, and this author. The project has been divided so that each part forms a separate master's thesis. This paper discusses the Very High Speed Integrated Circuit Hardware Description Language (VHDL) modeling of this processor. The asynchronous processor uses most of the R3000's instruction set and architectural features but differs in implementation. Three models, behavioral, dataflow, and structural, are constructed. The behavioral model describes the functionality of the R3000 without regard to implementation. The dataflow model represents the pipeline of the processor. It models the data flowing through the pipeline stages. The structural model represents the processor at the gate or structural level. The dataflow and structural model delay times were back annotated from circuit simulation runs. Each model is tested using a VHDL test bench to verify correct operation. Kevin Johnson and Scott Siers were responsible for the Very Large Scale Integration (VLSI) design and implementation. Kevin Johnson designed the register bank, the arithmetic logic unit (ALU), and the shifter. Scott Siers designed the bus control logic, the multiplier/divider unit (MDU), and the pipeline structure. This author also participated in the design of the asynchronous processor from a modeling perspective. The hardware designs were changed or completely redone depending upon the results of the VHDL modeling.

The main purpose of this project was to investigate the feasibility of converting an existing synchronous processor design to an asynchronous design. The MIPS R3000 was chosen for this task for numerous reasons. The R3000 has the best combination of instruction set size, architectural features, and system complexity to fully exploit the differences between synchronous and asynchronous design. The R3000 was one of the first reduced instruction set computer (RISC) processors and is a very simplistic, concise,

and elegant architecture. The number of instructions is minimal and there are only three addressing modes. For a 32-bit machine it has a small architecture and hence makes an ideal processor for thesis work. There is an abundance of literature written on the MIPS R3000. For example, the paper written by Asada, Okura, and Cho in 1992 [1] discusses the design of an asynchronous implementation of the MIPS data path. Another paper by Ginosar and Michell [2] discusses converting the MIPS pipeline using an asynchronous design methodology. Also, the book MIPS RISC Architecture [3] gives extremely low-level details of the synchronous version of the processor. These features make the R3000 an ideal processor to model.

The MIPS R3000 is a general purpose microprocessor that includes a 32-bit data path with thirty-two 32-bit general purpose registers. The R3000 has a 5-stage pipeline. The five stages are instruction fetch (IF), instruction decode (ID), arithmetic logic unit (ALU), memory (MEM), and register writeback (WB). The three main addressing modes are register, immediate, and jump. The R3000 is a reduced instruction set computer (RISC). One characteristic of a RISC based architecture is that it is a register based design. The processor only works on data contained in the registers. The order of operations is that data is loaded from memory into the registers, the processor works on the data in the registers, the result is stored back into a register, and finally the result is put back into memory. The advantage of this architecture is that the instruction set is smaller (reduced) and consists of simpler instructions. This allows the cycle time for each instruction to be short. All of the R3000's instructions are 32 bits in length. This is another advantage of RISC machines. The complexity of the instruction decoder is minimal and the instruction set size is limited.

Before any modeling was done, certain issues had to be considered. What architectural features were to be modeled? Which instructions were to be modeled? How many specific models should be designed and at what level of detail? During preliminary discussions on this thesis topic it was decided that the main architectural features of the

R3000 would be modeled: the 5-stage pipeline, the thirty-two 32-bit register bank, the hi/lo registers, and the 32-bit address and data paths. The features that would be left out are the coprocessor and coprocessor support, the cache for memory and instruction fetch, and the memory management. Since it was decided that memory would not be modeled, memory management would not be implemented. The coprocessor and memory management instructions were left out. Due to these architectural decisions and to limit the size of this thesis, the asynchronous processor instruction set was reduced.

The VHDL modeling in this thesis consists of the following three models: behavioral, dataflow, and structural. All models consist of three modules: memory, central processing unit (CPU), and compare. These three modules when put together with the test program and the expected results file create a complete test bench. Figure 1-1 shows the test bench setup. Each model will use the test bench along with test programs to verify correct operation. The memory and compare modules are relatively similar for all three models. The memory module is used as the main memory storage for the

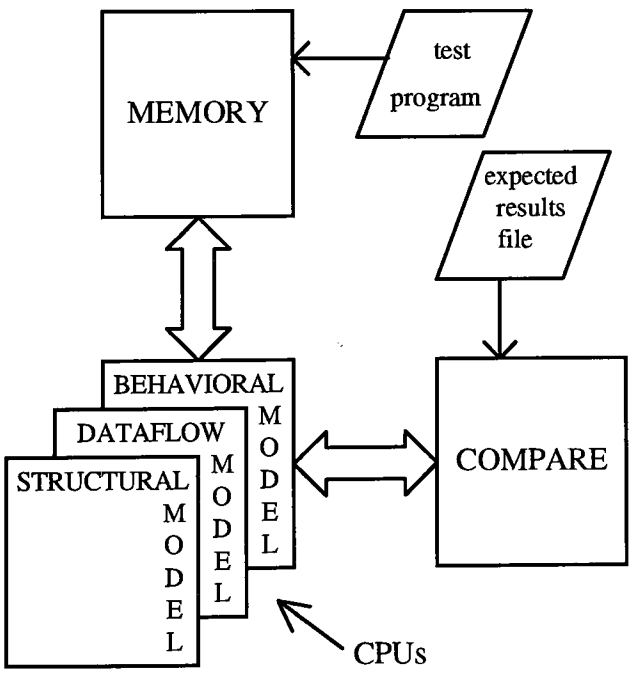


Figure 1-1. Test Bench Block Diagram

processor and holds the test program used in the test bench. The compare module is used to test the state of the processor after each instruction is executed and holds the expected results file. The CPU module corresponds to one of the three models.

Software was written for this thesis to assist in testing the models. An assembler, called MIPS Assembler (MASS), was written to convert MIPS assembly code into machine code that the models can understand. Also, an expected results program was written. This program, called MIPS Expected Results Assembler (MERA), allows the user to input the expected results data into a file. This file is loaded into the compare module and is tested against the state of the model after each instruction is executed. Another program was written for branch and jump instructions. This program, called FLOW, takes a program that contains branch and jump instructions and reorders the instructions to follow the program flow. The last program is called MIPS Preprocessor (MPP). This program takes the files created by MASS and MERA and copies them into two files that are used by the models. These two files, "machine" and "expected", are loaded by the memory and compare modules, respectively.

Six test programs were written for the models. Each of these tests correspond to a set of instructions. The program "ai.test" tests immediate arithmetic instructions. "ar.test" tests register arithmetic instructions. The third program, "jb.test", is used for jump and branch instructions. The program "ls.test" is used to test load and store instructions. The fifth test, "md.test", checks the multiplication and division instructions along with the move to and from the hi/lo registers. The last test file, "s.test", is used for the shift instructions.

The software tools used in this project are from Mentor Graphics Corporation and run on HP/Apollo Workstations. Five software tools were used: Design Architect, System-1076 (VHDL) editor and compiler, Quicksim II, Accusim, and IC Station. Design Architect is a schematic capture tool. The VHDL editor and compiler is incorporated into Design Architect. The digital simulator is Quicksim II. The VHDL simulator is

embedded into Quicksim II. Accusim is the analog circuit simulator. Finally, the mask layout editor is called IC Station.

2.0 CONCEPTS

One major feature of synchronous design is the use of a global clock. The very nature of synchronous design is to control all events based on this clock. Multiple events can happen but they will not be triggered until the next clock pulse. The order of events is of no concern. On the other hand, asynchronous design avoids the use of a global clock. Therefore, there is no convenient way to synchronize events. Here, the order of events is very important. A controller can be used between logic blocks as a communication device. The controller uses start and done signals as handshake signals. When one logic block is finished, it sends the controller a done signal. The controller can now send the next logic block a start signal. With synchronous design, the logic blocks can be tuned to start at a certain time based on the clock phase. However, the exact time at which a specific event starts or ends is not known in asynchronous design. This is why a controller is needed. It coordinates the timing of events through the use of handshaking signals.

2.1 ASYNCHRONOUS DESIGN

An asynchronous design approach was chosen over a synchronous one for many reasons. As transistor sizes in VLSI keep getting smaller, the major drawbacks to synchronous design become more apparent and difficult to tolerate. Two major disadvantages are the skew associated with a global clock and the increasing line delay that occurs when a signal is routed across a VLSI chip. One of the major design goals in synchronous design is to increase the performance of the processor by reducing the clock period. However, as the reduction in scale of VLSI systems continues, more and more of the clock period is used to account for clock skew. Global clock lines become more sensitive to loading and it becomes increasingly difficult to keep the various clock line signals in phase. The second issue involves line delay. In the past, the major delay in

circuit design was the transistor gate. Today, the line delay in signal routing is a major concern. To obtain substantial increases in performance, new architectures will have to be used to reduce the need for long metal lines. Circuit modules will have to be linked only by local interconnections and the modules will then communicate via self-timed handshaking schemes. The asynchronous design that eliminates the global clock and reduces the need and effects of long signal lines is an attractive solution in modern VLSI implementation.

2.2 HANDSHAKING CONTROL CIRCUIT

The handshaking control circuit (HCC) represents the control flow mechanism for the asynchronous machine. Every major logic block in the design has an HCC associated with it. The HCC synchronizes the events among the major logic blocks. The HCC coordinates control information from its own logic block and from other HCCs in the design. Figure 2-1 shows the HCC architectural organization. The major logic blocks shown in the diagram correspond to the five pipeline stages of the processor. An HCC is dependent on the previous and the next HCC. The HCCs on the ends of the pipeline are

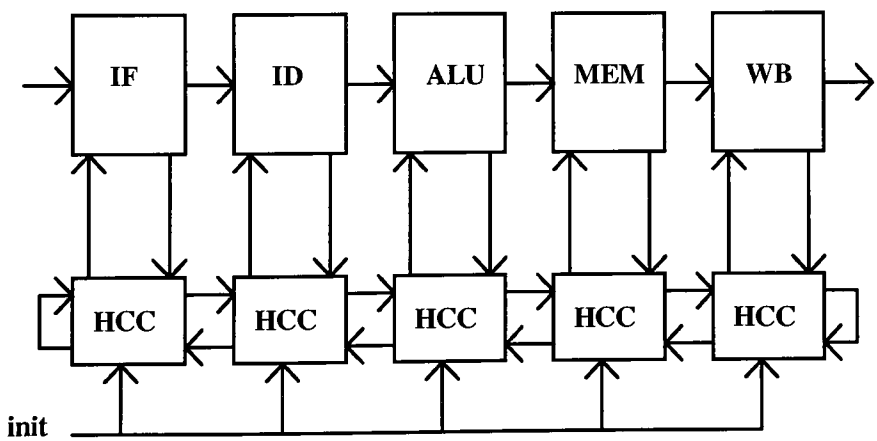


Figure 2-1. HCC Architectural Organization

only dependent on its one neighbor. For example, the ALU HCC is dependent on the ID and MEM HCCs. The ALU stage cannot begin operation until the ID stage is finished and the MEM stage has latched the previous data from the ALU stage. The HCC component block diagram is shown in Figure 2-2. The signal names and their descriptions are shown in Table 2-1.

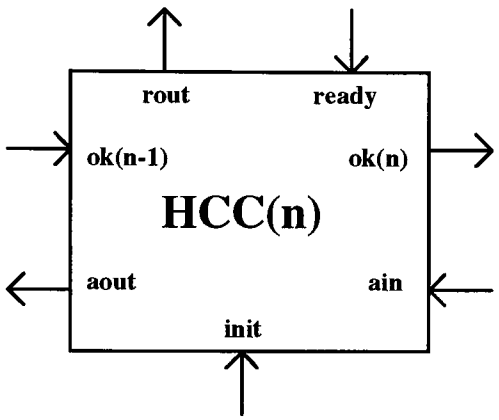


Figure 2-2. HCC Component Block Diagram

SIGNAL NAME	DESCRIPTION
init	Initializes the HCC
ok(n-1)	Previous stage has completed its operation
aout	Acknowledgment to previous stage that data has been latched
rout	Start signal to logic block
ready	Done signal from logic block
ok(n)	Signals next stage that present stage is completed
ain	Acknowledgment from next stage that data has been latched

Table 2-1. HCC Component Signal Names and Descriptions

The HCC schematic circuit diagram is shown in Figure 2-3 and its waveforms are shown in Figure 2-4. The operation of the HCC is as follows. The *init* signal is used to initialize the HCC and acts like a reset. The *ok(n-1)* signal is the start signal for the HCC. This signal comes from the previous HCC. When *ok(n-1)* goes low, it signals that the

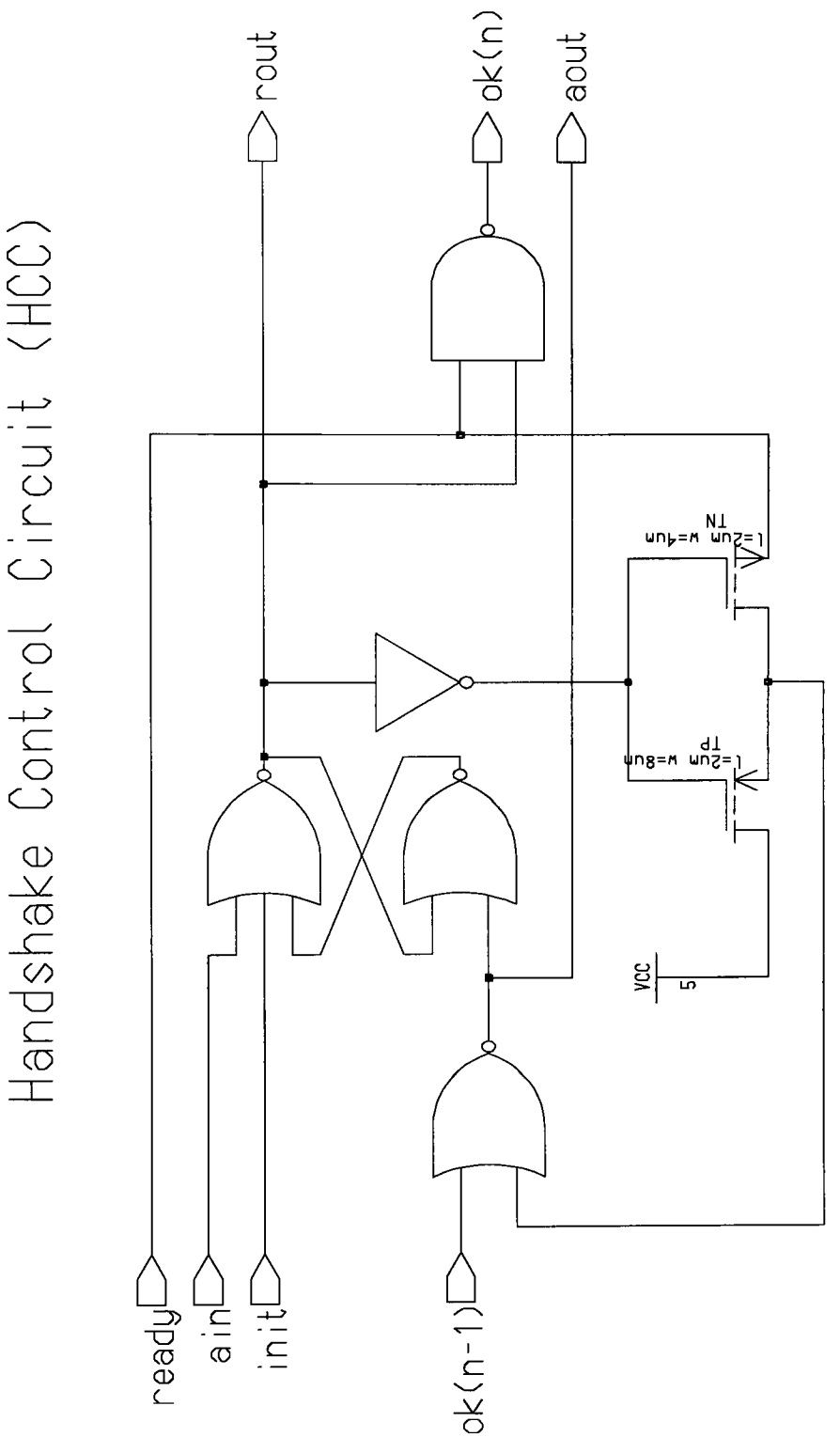


Figure 2-3. Handshaking Control Circuit (HCC) Schematic Diagram

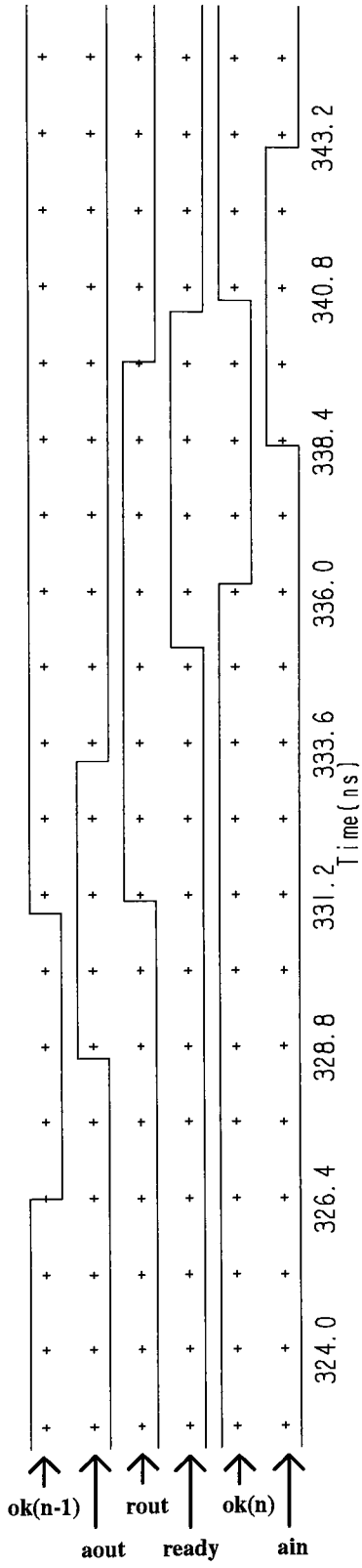


Figure 2-4. HCC Waveforms

previous stage has valid data. The HCC sends the acknowledgment signal *aout* high when *rout* and *ready* are both low. When *aout* is high and *ain* is low, *rout* goes high. *rout* is the start signal to the HCC's logic block. The HCC now waits for the completion of the logic block. This is signified by the *ready* line going high. Once *ready* is high, the HCC sends *ok(n)* low. This signals to the next HCC that the logic block has valid data. The HCC now waits for *ain* to go high which is an acknowledgment from the next HCC. *ain* going high causes *rout* to go low and *ok(n)* to go high. When *rout* goes low, *ready* goes low. The cycle then repeats itself.

2.3 VHDL

VHDL stands for VHSIC Hardware Description Language. VHSIC stands for Very High Speed Integrated Circuit. The Department of Defense (DOD) initiated the VHSIC program. The DOD also initiated VHDL to create a hardware description language that all VHSIC contractors could use to specify their designs. More importantly, this would allow designs to be transferred from one company to another and be totally independent of the tools and the platforms they run on.

VHDL is a high level language. High level constructs are important for design specification (behavioral model) and testing. VHDL allows the user to concentrate on the behavioral aspects of the design and forget the low level details during the beginning stages of design. VHDL is an IEEE standard formalized in specification 1076-1987 and updated in specification 1076-1992. Writing in VHDL allows the user to port the source code over to another hardware platform with ease. All that needs to be done is to recompile the source code on the new platform and to run the new platform's simulator.

VHDL is a programming language that can simulate concurrent events. This allows the user to specify multiple events at the same simulation time. VHDL uses the

concept of delta delay, which is an infinitesimally small delay, to order events that occur at the same simulation time. VHDL software tools incorporate a simulator to test the design, and a test bench can be used to simplify the testing process. Also, VHDL has packages specifically designed to model hardware at different stages of design.

2.4 TOP DOWN DESIGN

Top down design is a popular design methodology. It guides a design from a high level to a low level of abstraction. The system's functionality is described at a high level of abstraction. Implementation of this functionality is not an issue at this level. On the contrary, the system's low level details are only considered at a low level of abstraction. The low level describes the gate level implementation. As the design progresses from one level of abstraction to the next, functionality of the design is completed and set at the higher levels and then the implementation of this functionality is created at the lower levels. This thesis uses this concept by using the three different models. Each model is another level of abstraction.

A behavioral model is a model that describes the behavior of the hardware entity under test. An entity is a hardware unit that can be as simple as a gate or as complex as an entire electronic system. A behavioral model does not explicitly specify the structure of the entity but specifies its functionality. Another way of looking at the behavioral style of modeling is the well known "black box" approach. The hardware unit is described in terms of its input-output mapping without specifying the model's technology, components, or dataflow.

A behavioral model projects a very high level of abstraction. At the first stages of design, the behavioral model relieves the user of the low-level details of the design and implementation of the entity. This frees the designer and/or user to concentrate on the

behavior of the system in question. Overall, the behavioral model provides a means to better understand the functionality of the entity. Finally, due to the high level of abstraction, the behavioral model executes much faster than other modeling schemes. This is advantageous at the beginning stages of a design when many simulations need to be done.

The behavioral modeling technique was used to model the functionality of the R3000 without regard to hardware implementation. The behavioral model was used to understand the operation of every instruction that was implemented. The entire behavioral model is actually one VHDL PROCESS statement. All instructions in a PROCESS statement are executed sequentially. Therefore, the pipeline was modeled in a sequential fashion. An instruction is fetched, decoded, executed, stored, and tested all within one cycle of the PROCESS statement. The next instruction is not worked on until the first instruction is finished. The data type used in this model is the BIT VECTOR. This choice is discussed in section 2.4. The computation instructions are implemented using functions and procedures since this model is only concerned with the operation of the processor and not on how it is implemented. These functions and procedures are located in a VHDL PACKAGE.

A dataflow model describes the behavior of the hardware entity just like the behavioral model but in more detail. Dataflow modeling involves some implementation details since it is concerned with the flow of data from one part to another. The dataflow model starts to break the functionality of the behavioral model down into compartments. An example of this is the breakdown of the pipeline. Each stage of the pipeline performs a different function. The dataflow model specifies how the data will flow from one section to another.

Dataflow modeling is similar to behavioral modeling since there are no gates to specify low level implementation, but it is also similar to structural modeling because of the use of components (even though the components are written using the dataflow style).

The dataflow model uses the same data types and package of declarations, functions, and procedures as the behavioral model. This was done to simplify the model. The main focus of attention was to get the pipeline stages talking to each other and to assure that each stage was decoding and working on the proper instruction. Also, to keep the dataflow model simple, only two logic states (0 and 1) were used to describe and simulate the model.

The dataflow model provided the next level of abstraction by modeling the pipeline stages concurrently. This model is used to design the handshaking interface protocol between the pipeline stages. The main objective of this model was to establish proper and efficient communication between the pipeline stages. A rough protocol would be designed and then tested using the dataflow model. Using the simulation waveform outputs of the model, the protocol was modified to improve the design. Also, design errors were corrected using the dataflow output waveforms. This continued throughout the entire design cycle of the asynchronous processor.

The dataflow model is largely a hierarchical structure of dataflow components. The top level component is the CPU. The CPU is then broken up into eight unique components: the five pipeline stages, the HCC, the bus controller (BC), and the exception handler (EH). Each stage of the pipeline has an HCC associated with it. Each of these components are made of other smaller components and primitives. Examples of the primitives are multiplexers, latches, and edge detectors. The components were used to speed up the model building process. When a particular component is needed, the proper code is called through use of the VHDL COMPONENT instantiation and PORT MAP statements.

The original dataflow model was constructed using arbitrary delay times just to get the model working. Back annotation was used once the model was completed, tested, and verified for correct operation. The new delay times were obtained from Accusim simulation runs of circuit descriptions of the various components and pipeline stages.

These simulations were performed and discussed in Scott Siers' thesis and Kevin Johnson's thesis "Design and Implementation of an Asynchronous Version of the MIPS R3000 Microprocessor" [10,19].

The structural model represents the processor at its gate or structural level. This is the lowest form of abstraction. It is the lowest and most detailed level of description. The structural model uses a set of components connected by signals. With the structural model, the behavior of the entity is not apparent from the model. This is unlike the behavioral model where the behavior or functionality is readily apparent. Component instantiation is the major VHDL device that facilitates a structural or gate level nature.

2.5 DATA TYPES

Originally, integers were chosen as the main modeling data type. However, this presented some problems. The representation of data by integers appears to be ideal but instructions need a more robust data type. Different pieces of the instruction represent different aspects and conditions of the computer. An instruction holds operation, register operand, and memory address information. Also, depending on the addressing mode, an instruction holds different types and amounts of information. A composite type called a record was considered in order to hold all the different fields of information contained in an instruction. However, Mentor Graphic's version of VHDL now in use (version 8.1) does not support records.

At this time, a decision was made to use a data type called a bit vector. Bit vectors are ideal to use because they represent the language that the computer understands. Bit vectors can be manipulated using two methods. Using the first method, the bit vector is converted to an integer, the integer is operated on, and then the integer is converted back to a bit vector. This method is faster to execute but involves much conversion. The

second method is to manipulate the individual bits. These bit manipulations are handled by procedures and functions. An example of this is adding two bit vectors together. The two bit vectors are passed to the overloaded "+" function. The "+" function adds the individual bits together and returns a bit vector result. These specialized functions add complexity and slow execution time.

3.0 BEHAVIORAL MODEL

The behavioral model is an instantiated component. The name of the model is CPU, since the central processing unit is the main part of the MIPS R3000 that is modeled. This CPU component is part of a test bench shown in Figure 3-1. The other two components of the test bench are the memory and compare modules.

The memory module is used as main memory storage for the processor. It is accessed by six interface signals. The memory control signals, *mem_control_sig* and

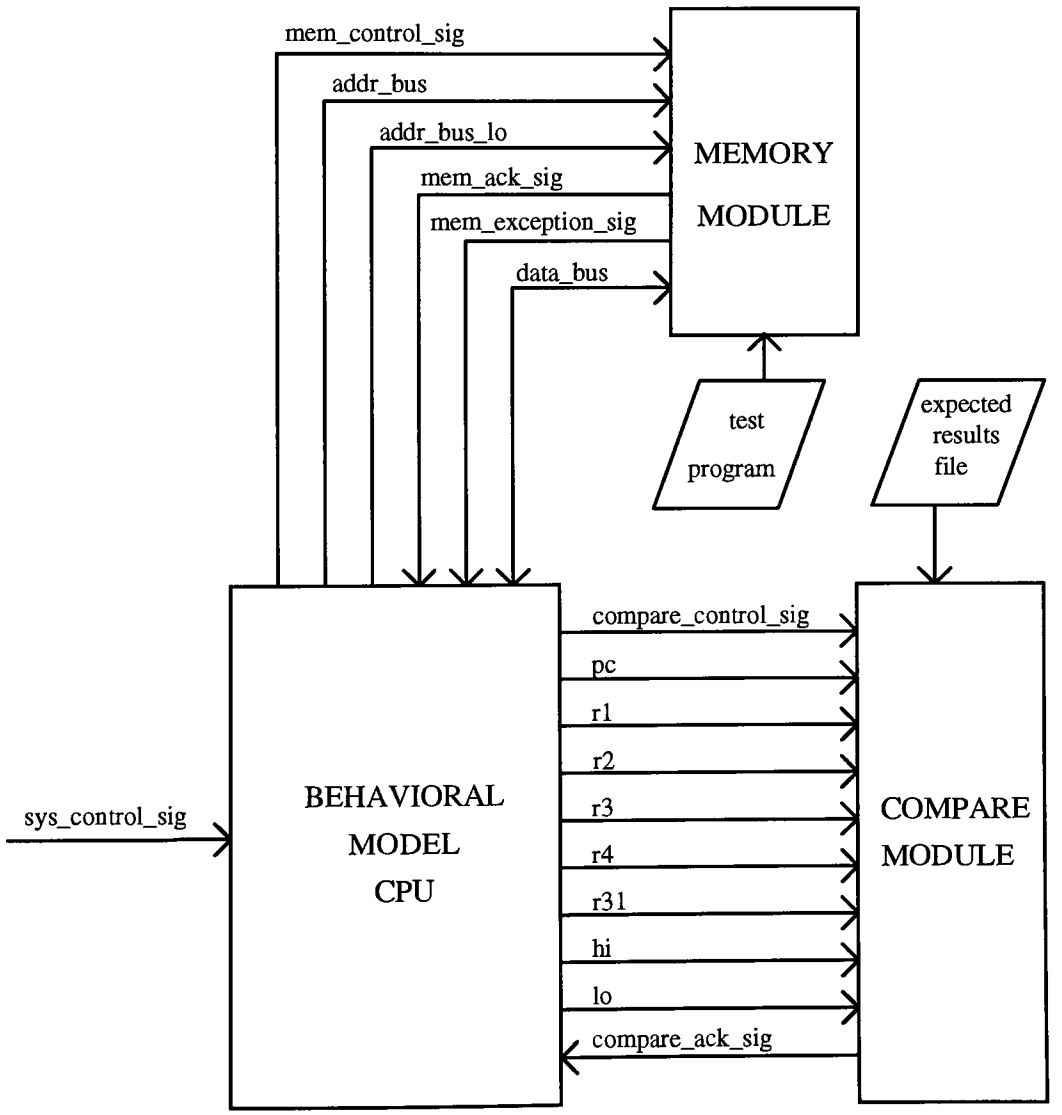


Figure 3-1. Behavioral Model Test Bench

mem_ack_sig, provide a fully interlocked handshaking protocol between the CPU component and the memory component. The address bus is broken into two separate signals: *addr_bus* and *addr_bus_lo*. *Addr_bus* provides the upper 30 bits (bits 2 through 31) of the 32 bit address bus, which accesses a word of memory (a word of memory is 32 bits wide). *Addr_bus_lo* provides the lower two bits (bits 0 and 1) of the address bus, which accesses a byte of memory (a byte of memory is 8 bits wide). The *data_bus* is a 32-bit bi-directional bus. It is used to transfer data between the CPU and the memory module. Finally, the exception control signal, *mem_exception_sig*, is activated when a memory exception occurs.

The compare module tests the state of the processor after each instruction is executed. It is accessed by ten interface signals. The compare control signals, *compare_control_sig* and *compare_ack_sig*, provide the handshaking between the CPU and the compare module. The *pc* signal monitors the CPU program counter. Signals *r1, r2, r3, r4,* and *r31* monitor the contents of the specified registers. The *hi* and *lo* signals monitor the multiplication and division storage registers.

The CPU module or component uses a VHDL PROCESS to model the processor. A PROCESS statement is a collection of sequential statements that describe the functionality or behavior of a portion of an ENTITY. A PROCESS is first entered during the initialization phase of a simulation. During this initialization, it continues to execute until it suspends due to an explicit WAIT statement or an implicit WAIT due to a sensitivity list. Also, once a PROCESS is entered, it is never exited. It is always in one of two states: active or suspended. A PROCESS is active when it is executing and suspended when it is waiting for a certain event to occur.

A PROCESS is sensitive to signals in a sensitivity list. If an event occurs on any one or more of the signals in the sensitivity list, the PROCESS is executed. The statements in the PROCESS are executed in a sequential fashion. It suspends after executing the last sequential statement and waits for another event to occur on a signal in

the sensitivity list. A WAIT statement can be used in place of a sensitivity list. A PROCESS executes until a WAIT statement is reached. The PROCESS is suspended until an event occurs on the signal in the WAIT statement.

The body outline of the PROCESS used in this model is shown in Figure 3-2. It is sensitive to the *sys_control_sig* signal using a WAIT statement. The process is broken down, using a VHDL CASE statement, into four sections each corresponding to the four modes of operation: *stop*, *reset*, *load*, and *run*.

```
processor: PROCESS
    -- process declarations
BEGIN
      WAIT ON sys_control_sig;
      CASE sys_control_sig IS
        WHEN stop =>
             .
             .
        WHEN reset =>
             .
             .
        WHEN load =>
             .
             .
        WHEN run =>
             .
             .
      END CASE;
END PROCESS processor;
```

Figure 3-2. The Body Outline of the Processor Process

The *load* mode, shown in Figure 3-3, is part of the system initialization. *load* sends signals to the memory and compare components to load the system programs. More precisely, the *load* mode initiates handshake signals with the memory and compare components. It sends the *load* signal to the memory component via *mem_control_sig* and to the compare component via *compare_control_sig*. When both components are finished loading their respective programs, they send back their acknowledgment signals. A reset signal is then sent to both components.

```
WHEN load =>
  run_mode_flag := no;

  mem_control_sig <= load AFTER delay;
  WAIT UNTIL mem_ack_sig = yes;
  mem_control_sig <= reset AFTER delay;
  WAIT UNTIL mem_ack_sig = no;

  compare_control_sig <= load AFTER delay;
  WAIT UNTIL compare_ack_sig = yes;
  compare_control_sig <= reset AFTER delay;
  WAIT UNTIL compare_ack_sig = no;
```

Figure 3-3. Load Mode

Run mode, shown in Figure 3-4, is the largest part of the *processor* PROCESS. It starts by first setting *run_mode_flag* to *yes*. This flag controls the exit condition from a VHDL WHILE loop. While *run_mode_flag* is set to a value of *yes*, the processor continues to work as it cycles through the WHILE loop. When *run_mode_flag* changes to *no*, the run mode WHILE loop is exited and the processor ceases to work. The main part of the processor model is contained inside this WHILE loop. The WHILE loop is broken down into nine sections: instruction fetch (IF), instruction decode (ID), instruction execution (IE), exception handling, program counter update, memory latency handling, branch delay update, signal update, and testing.

```
WHEN run =>
  run_mode_flag := yes;
  WHILE run_mode_flag = yes LOOP

      -- instruction fetch
      -- instruction decode
      -- determine which type of addressing it is
      -- set fields accordingly
      -- exception handling
      -- delay slot/pc increment
      -- latency of one instruction on register load
      -- set branch delay flag
      -- update signals
      -- compare machine state with expected results

  END LOOP;
```

Figure 3-4. Run Mode While-Loop Shell

3.1 INSTRUCTION FETCH

The instruction fetch portion of the code is very simple; it is only one line and can be seen in Figure 3-5. Instruction fetch is accomplished by calling the *mem_read* procedure. The *mem_read* procedure is discussed in the next section. The two arguments needed to do a memory read are the starting address of the data to read and the size of the data. The value in the program counter register (*pc_reg*) gives the starting address and the value *word* gives the data size. The data size is of type *word* since all instructions are 32-bits long. The procedure result is placed in *current_inst*, a 32-bit variable which stands for current instruction.

```
-- fetch next instruction
mem_read(pc_reg, word, current_inst);
```

Figure 3-5. Instruction Fetch

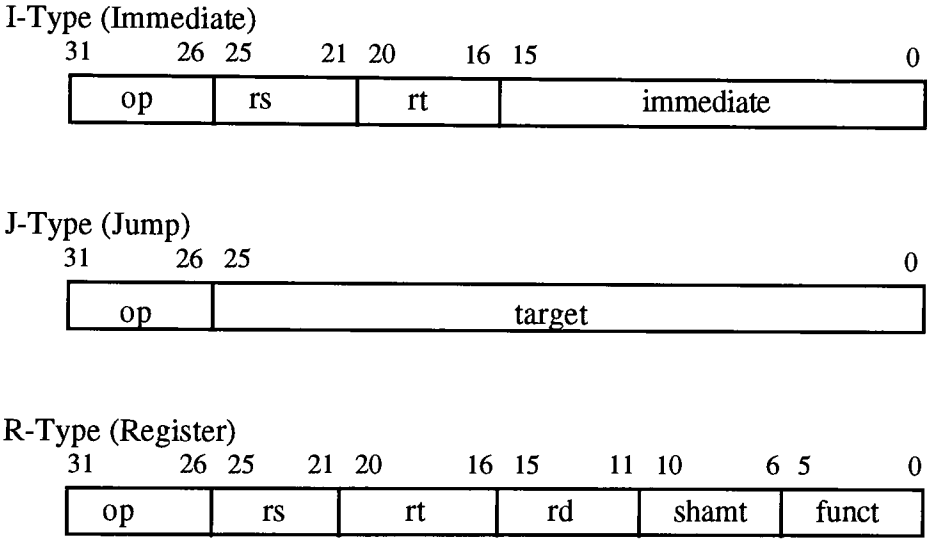
mem_read, shown in Figure 3-6, takes two arguments, the memory address and the memory size, and returns the result from the data bus. *mem_read* first sets up the address bus signals, *addr_bus* and *addr_bus_lo*, from the value passed to it. Depending upon what type of memory operation is to be performed, it selects from the *size* variable what value to assign to *mem_control_sig*. It now waits for the memory component to send back the memory acknowledge signal, *mem_ack_sig*. The data from the memory is now ready to be transferred to *result*. *mem_read* now sends a reset signal to the memory and waits again for an acknowledgment. This completes the fully interlocked handshaking scheme. Finally, the *mem_exception_sig* is checked to see if a memory exception has occurred.

```
PROCEDURE mem_read(addr: IN bit_32; size: IN size_type;
                   result: OUT bit_32) IS
BEGIN
    addr_bus <= addr(31 DOWNTO 2) AFTER delay;
    addr_bus_lo <= addr(1 DOWNTO 0) AFTER delay;
    CASE size IS
      WHEN byte =>
        mem_control_sig <= read_b AFTER delay;
      WHEN ubyte =>
        mem_control_sig <= read_ub AFTER delay;
      WHEN halfword =>
        mem_control_sig <= read_hw AFTER delay;
      WHEN uhalfword =>
        mem_control_sig <= read_uhw AFTER delay;
      WHEN word =>
        mem_control_sig <= read_w AFTER delay;
      WHEN lefty =>
        mem_control_sig <= read_l AFTER delay;
      WHEN righty =>
        mem_control_sig <= read_r AFTER delay;
    END CASE;
    WAIT UNTIL mem_ack_sig = yes;
    result := data_bus;
    mem_control_sig <= reset AFTER delay;
    WAIT UNTIL mem_ack_sig = no;
    IF mem_exception_sig = yes THEN
        exception_flag := addr_load;
    END IF;
END mem_read;
```

Figure 3-6. Memory Read Procedure in Processor Process

3.2 INSTRUCTION DECODE

Instruction decode involves extracting the op-code from the current instruction, determining what instruction grouping the instruction belongs to, and then setting the operand and address fields accordingly. The next instruction to execute is stored in the *current_inst* variable. An instruction can take on one of the three instruction formats shown in Figure 3-7. The major op-code is the six most significant bits of the instruction. On some instructions, a minor op-code is used. This minor op-code, also called a function field, is the least significant six bits of some instructions.



where:

op	is a 6-bit operation code
rs	is a 5-bit source register specifier
rt	is a 5-bit target (source/destination) register or branch condition
immediate	is a 16-bit immediate, branch displacement or address displacement
target	is a 26-bit jump target address
rd	is a 5-bit destination register specifier
shamt	is a 5-bit shift amount
funct	is a 6-bit function field

Figure 3-7. R3000 Instruction Formats

To understand instruction decoding, the R3000 instruction op-code bit encoding needs to be addressed. This bit encoding is shown in Table 3-1. Table 3-1 is composed of three tables: *opcode*, *special*, and *bcond*. The *opcode* table displays all the possible bit combinations of the major op-code. The rows of the *opcode* table represent the three most significant bits of the major op-code field. The columns represent the three least significant bits. When the major op-code of an instruction is equal to 00 octal, then the instruction is a *special* instruction. The *special* instructions are shown in the *special* table. The *special* instructions are encoded through the minor op-code field variable called *funct* (function field). The *special* table displays all the possible combinations of the *special* instructions. The rows of the *special* table represent the three most significant bits of *funct*. The columns represent the three least significant bits. When the major op-code is equal to 01 octal, then the instruction is a *branch conditional (bcond)* instruction. The *branch conditional* instructions are shown in the *bcond* table of Table 3-1. These instructions are encoded through a *bcond* field variable called *reg_funct* which is a five bit field. The rows of the *bcond* table represent the two most significant bits of the *reg_funct* field. The columns represent the three least significant bits of *reg_funct*.

OPCODE

31..29 \ 28..26	0	1	2	3	4	5	6	7
0	special	bcond	j	jal	beq	bne	blez	bgtz
1	addi	addiu	slti	sltiu	andi	ori	xori	lui
2	cop0 **	cop1 **	cop2 **	cop3 **	*	*	*	*
3	*	*	*	*	*	*	*	*
4	lb	lh	lwl	lw	lbu	lhu	lwr	*
5	sb	sh	swl	sw	*	*	swr	*
6	lwc0 **	lwc1 **	lwc2 **	lwc3 **	*	*	*	*
7	swc0 **	swc1 **	swc2 **	swc3 **	*	*	*	*

SPECIAL

5..3 \ 2..0	0	1	2	3	4	5	6	7
0	sll	*	srl	sra	sllv	*	srlv	srav
1	jr	jalr	*	*	syscall	break	*	*
2	mfhi	mthi	mflo	mtlo	*	*	*	*
3	mult	multu	div	divu	*	*	*	*
4	add	addu	sub	subu	and	or	xor	nor
5	*	*	slt	sltu	*	*	*	*
6	*	*	*	*	*	*	*	*
7	*	*	*	*	*	*	*	*

BCOND

20..19 \ 18..16	0	1	2	3	4	5	6	7
0	bltz	bgez						
1								
2	bltzal	bgezal						
3								

* Operation codes marked with an asterisk cause reserved instruction exceptions and are reserved for future versions of the architecture.

** Operation codes marked with two asterisks are not implemented in the asynchronous version.

Table 3-1. R3000 Instruction Opcode Bit Encoding

The op-code is extracted from *current_inst* by using a bit-vector array range, as shown in Figure 3-8. The top six bits of current_inst is extracted and stored in a variable called *opcode* using an array range (a bit-vector is an array of bits) from bit 31 down to bit 26. The op-code is further broken down by extracting a segment of the variable *opcode*. This is stored in a variable called *opcode_seg*. *Opcode_seg* is the three most significant bits of *opcode* and is used to determine the instruction grouping.

```
opcode := current_inst(31 DOWNTO 26);
opcode_seg := opcode(5 DOWNTO 3);
```

Figure 3-8. Extracting the Op-code from the Instruction

After the op-code is extracted from the current instruction, the instruction grouping is determined. The major op-code instructions are broken up into the following groups: *special, bcond, jump, branch, immediate, load,* and *store* instructions. A VHDL IF-ELSIF-ELSE statement, shown in Figure 3-9, is used to direct the program flow to the correct instruction grouping. All instructions are named the same as the R3000 instruction set except for the addition of the letter "i" and an underscore bar ("i_"). The R3000 major op-code instruction *special* is called *i_special*, for example. All the instructions are defined in a VHDL PACKAGE as constants to improve code readability and debugging. The *i_special* instruction is defined as a constant bit value of "000000", for example.

The IF-ELSIF-ELSE statement is broken up into nine branches. When the major op-code is equal to the constant *i_special* then the first branch is taken. This first branch handles all the R3000 special instructions. The second branch handles the four *branch conditional* instructions. This occurs when the major op-code is equal to the constant *i_bcond*. The two *jump* instructions, *i_j* and *i_jal*, are found in the third branch. The fourth branch holds the *branch* instructions and is taken when *opcode_seg* is equal to "000". The fifth branch holds the *ALU immediate* instructions and is taken when

opcode_seg is equal to "001". The *load* and *store* instructions are grouped together in the sixth branch. The seventh branch is the *halt* instruction. This instruction is used to stop the processor and is used at the end of every test program. It is not part of the R3000 instruction set. The eighth and ninth branches handle instructions *not-implemented* and *reserved* instructions, respectively. Each of the branches will be discussed in more detail in the following sections.

```
-- special instructions
IF opcode = i_special THEN
.
.
-- conditional branch instructions
ELSIF opcode = i_bcond THEN
.
.
-- jump, jump and link instructions
ELSIF opcode = i_j OR opcode = i_jal THEN
.
.
-- branch instructions
ELSIF opcode_seg = b"000" THEN
.
.
-- ALU immediate instructions
ELSIF opcode_seg = b"001" THEN
.
.
-- load and store instructions
ELSIF opcode_seg = b"100" OR opcode_seg = b"101" THEN
.
.
-- halt instruction (not a mips instruction)
ELSIF opcode = i_halt THEN
.
.
-- instructions not implemented
ELSIF (opcode_seg = b"010" OR opcode_seg = b"110" OR
       opcode_seg = b"111") AND opcode(2) = '0' THEN
.
-- reserved instruction
ELSE
.
END IF;
```

Figure 3-9. IF-ELSIF-ELSE Statement used for Instruction Decode

The *special* instructions branch of the IF-ELSIF-ELSE statement is shown in Figure 3-10. It first extracts the fields from the current instruction. All *special* instructions use the register instruction format. This format consists of the minor op-code field (this is also known as the function field), the source register field (*rs*), the target register field (*rt*), the destination register field (*rd*), and the shift amount field (*shamt*). The minor op-code is calculated from the current instruction by extracting an array range on *current_inst* and storing it in *funct*. The register operand fields (*rs, rt, rd,* and *shamt*) are more complicated. First, the proper array range of *current_inst* is extracted. Next, this extracted array of bits is converted to a natural number using the bits-to-natural (*bton*) function. Lastly, the natural number, which specifies a register number from 0 to 31, is stored in the appropriate variable. Once all the fields are set accordingly, the program flow branches to the proper instruction. A VHDL CASE statement is used to select which *special* instruction is executed. The specific *special* instruction is denoted by the minor op-code which is stored in the *funct* field.

```
IF opcode = i_special THEN

    funct := current_inst(5 DOWNTO 0);
    rs    := bton(current_inst(25 DOWNTO 21));
    rt    := bton(current_inst(20 DOWNTO 16));
    rd    := bton(current_inst(15 DOWNTO 11));
    shamt := bton(current_inst(10 DOWNTO 6));

    CASE funct IS
      WHEN i_sll =>
       .
       .
      WHEN i_srl =>
       .
       .
      WHEN i_sra =>
       .
       .
      WHEN i_sllv =>
       .
       .
       .
    END CASE;
```

Figure 3-10. Special Instruction Branch of IF-ELSIF-ELSE Statement

The *branch conditional* (*bcond*) instructions of the IF-ELSIF-ELSE statement is shown in Figure 3-11. The *bcond* instructions use the immediate instruction format. Therefore, the fields that need to be set are the register source (*rs*), *bcond* function (*reg_funct*), and offset (*offset*). The *bcond* function, which is the *branch conditional* minor op-code, is stored in the *reg_funct* variable. The variable is called *reg_funct* because the minor op-code information field is located in the same place as the target register field for other instruction formats in the *current_inst* variable. The offset is a 16-bit value that is extracted from the lower 16 bits of the current instruction. The minor op-code is calculated from the current instruction by extracting an array range on *current_inst* and storing it in *reg_funct.* The *rs* field is calculated the same way as the *rs* field in the special instructions section. The array range is extracted from *current_inst.* The array range is then converted to a natural number using the *bton* function. Lastly, the value is stored in the *rs* field variable. Once all the fields are set accordingly, the program branches to the proper branch conditional instruction. This is calculated using a CASE statement and selecting on *reg_funct.*

```
ELSIF opcode = i_bcond THEN

    rs        := bton(current_inst(25 DOWNTO 21));
    reg_funct := current_inst(20 DOWNTO 16);
    offset    := current_inst(15 DOWNTO 0);

    CASE reg_funct IS
      WHEN i_bltz =>
        .
        .
      WHEN i_bgez =>
        .
        .
      WHEN i_bltzal =>
        .
        .
      WHEN i_bgezal =>
        .
        .
    END CASE;
```

Figure 3-11. Bcond Instruction Branch of IF-ELSIF-ELSE Statement

The *jump* instructions branch of the IF-ELSIF-ELSE statement is shown in Figure 3-12. The *jump* instructions use the jump instruction format. This format uses a *target* field. The *target* field is a 26-bit jump target address and is extracted from the lower 26 bits of *current_inst*. Once the *target* field is set, the program branches to the proper jump instruction. This is calculated by using a CASE statement and selecting on *opcode*.

```
ELSIF opcode = i_j OR opcode = i_jal THEN
    target := current_inst(25 DOWNTO 0);
    CASE opcode IS
      WHEN i_j =>
        .
        .
      WHEN i_jal =>
        .
        .
    END CASE;
```

Figure 3-12. Jump Instruction Branch of IF-ELSIF-ELSE Statement

The *branch* instructions branch of the IF-ELSIF-ELSE statement is shown in Figure 3-13. The *branch* instructions use the immediate instruction format. Therefore, the fields that need to be set are *rs*, *rt*, and *offset*. Both register operand fields, *rs* and *rt*, are first extracted from *current_inst*, converted using the *bton* function, and finally stored in their respective variables. The *offset* value is extracted from *current_inst* and stored in the *offset* variable. Once these three fields are set, the program branches to the proper *branch* instruction. This is calculated using a CASE statement and selecting on *opcode*.

```
ELSIF opcode_seg = b"000" THEN

    rs     := bton(current_inst(25 DOWNTO 21));
    rt     := bton(current_inst(20 DOWNTO 16));
    offset := current_inst(15 DOWNTO 0);

    CASE opcode IS
      WHEN i_beq =>
      .
      .
      WHEN i_bne =>
      .
      .
      WHEN i_blez =>
      .
      .
      WHEN i_bgtz =>
      .
      .
    END CASE;
```

Figure 3-13. Branch Instruction Branch of IF-ELSIF-ELSE Statement

The *ALU immediate* instructions branch of the IF-ELSIF-ELSE statement is shown in Figure 3-14. As the name implies, the *ALU immediate* instructions use the immediate instruction format. The three fields that are set in this instruction format are *rs*, *rt*, and *immed*. As stated before, *rs* stands for register source and *rt* stands for register target. *Immed* is a variable used for the immediate field. The immediate field is a 16-bit immediate, branch, or address displacement. The *rs* and *rt* fields are extracted, converted, and stored in their respective variables. The immediate field is extracted from *current_inst* and stored in *immed*. Once the fields are set, the program branches to the proper *ALU immediate* instruction. This is done using a CASE statement with *opcode* as the selector.

```
ELSIF opcode_seg = b"001" THEN

    rs := bton(current_inst(25 DOWNTO 21));
    rt := bton(current_inst(20 DOWNTO 16));
    immed := current_inst(15 DOWNTO 0);

    CASE opcode IS
      WHEN i_addi =>
        .
        .
      WHEN i_addiu =>
        .
        .
      WHEN i_slti =>
        .
        .
        .
        .
    END CASE;
```

Figure 3-14. ALU Immediate Instruction Branch of IF-ELSIF-ELSE Statement

The *load* and *store* instructions branch of the IF-ELSIF-ELSE statement is shown in Figure 3-15. The *load* and *store* instructions use the immediate instruction format with the following three fields: *rt*, *base*, and *offset*. *rt* stands for the target register, *base* is an alias for the *rs* field, and *offset* is an alias for the immediate field. Once all the fields are extracted, converted, and stored, the program branches to the proper *load* or *store* instruction depending on the value of *opcode*.

```
ELSIF opcode_seg = b"100" OR opcode_seg = b"101" THEN

    base := bton(current_inst(25 DOWNTO 21));
    rt := bton(current_inst(20 DOWNTO 16));
    offset := current_inst(15 DOWNTO 0);

    CASE opcode IS
      WHEN i_lb =>
      .
      .
      WHEN i_lh =>
      .
      .
      .
      .
      WHEN i_sb =>
      .
      .
      WHEN i_sh =>
      .
      .
      .
      .
    END CASE;
```

Figure 3-15. Load and Store Instruction Branch of IF-ELSIF-ELSE Statement

The *halt* instruction branch is shown in Figure 3-16. This instruction is not part of the R3000 instruction set. It is added to aid testing and debugging. The *halt* instruction is added at the end of every test program. It stops the processor by setting the *run_mode_flag* to *no*. Note, the *inst* signal is used to monitor the state of the current instruction with the VHDL simulator.

```
ELSIF opcode = i_halt THEN
    inst <= op_halt;
    run_mode_flag := no;
```

Figure 3-16. Halt Instruction Branch of IF-ELSIF-ELSE Statement

The *not-implemented* instruction branch is shown in Figure 3-17. These instructions are found in the R3000 instruction set but are not implemented in the models. As shown in the figure, the *inst* and *exception_flag* signals are set to *not_implmt*.

```
ELSIF (opcode_seg = b"010" OR opcode_seg = b"110" OR
       opcode_seg = b"111") AND opcode(2) = '0' THEN
    inst <= not_implmt;
    exception_flag := not_implmt;
```

Figure 3-17. Not Implemented Instruction Branch of IF-ELSIF-ELSE Statement

The last instruction branch of the IF-ELSIF-ELSE statement is used for *reserved* instructions and is shown in Figure 3-18. These op-code values cause reserved instruction exceptions and are reserved for future versions of the architecture by the manufacturer. The *inst* signal is set to *reserved* and the *exception_flag* signal is set to the *reserved_inst* exception.

```
ELSE
    inst <= reserved;
    exception_flag := reserved_inst;
```

Figure 3-18. Reserved Instruction Branch of IF-ELSIF-ELSE Statement

3.3 INSTRUCTION EXECUTION

The instructions are divided into the following six groups: *special, branch conditional, jump, branch, ALU immediate,* and *load/store* instructions. The groups are discussed separately with a few examples for each group. This instruction grouping follows the instruction op-code bit encoding chart previously shown in Table 3-1. Note, every instruction uses the *inst* signal to display which instruction is currently being executed. Also, since register 0 (*r0*) is hard wired to the value zero, every instruction that stores a value in a register has to check that the destination register is not zero. If the destination register is zero, then the instruction is ignored. Lastly, many instructions perform their operations by calling a function that is located in a VHDL PACKAGE.

SPECIAL INSTRUCTIONS

The *special* instructions are further divided into the following groups: *shift, jump register, special, multiply/divide,* and *3-operand register* instructions. There are six shift instructions: *shift-left logical (sll), shift-right logical (srl), shift-right arithmetic (sra), shift-left logical variable (sllv), shift-right logical variable (srlv),* and *shift-right arithmetic variable (srav).* The logical shift instructions insert zeroes into the vacant bit positions. If the shift is to the left, zeroes are inserted into the low order bits. A right shift inserts zeroes into the high order bits. An arithmetic shift uses sign extension when inserting values into the vacant bit positions. The arithmetic shift is only used on a right shift. Therefore when a right arithmetic shift is performed, the high order bits are sign extended. A variable shift gets its shift amount from the contents of a register. More precisely, the low order 5 bits of register *rs* specify the number of bits to shift. If the shift is not variable, then the shift amount is held in the *shamt* field of those particular shift instructions. As an example, the *sll* instruction is shown in Figure 3-19. *sll* performs the

shift by calling the *shift_ll* function. *Shift_ll* takes two arguments: a 32-bit value to shift and the shift amount. *shift_ll* returns a 32-bit shifted value. More precisely, *sll* shifts the contents of register *rt* left by *shamt* bits, inserting zeroes into the low order bits. It then places the 32-bit result in register *rd*.

```
WHEN i_sll =>
  inst <= op_sll;
  IF rd /= 0 THEN
     reg(rd) := shift_ll(reg(rt), shamt);
  END IF;
```

Figure 3-19. The Shift Left Logical Instruction

There are two *jump register* instructions: *jump register (jr)* and *jump and link register (jalr)*. Both instructions jump to an address contained in register *rs* with a one instruction delay. The *jalr* instruction also places the address of instruction following the delay slot in register *rd*. As an example, the *jr* instruction is shown in Figure 3-20. The address that is stored in register *rs* is the location to which the program jumps after a delay of one instruction. Since the instruction in the delay slot needs to be executed before the jump is executed, the address is stored in a temporary variable called *pc_temp*. At the proper time, the program counter (PC) is loaded with the value in *pc_temp*. The *delay_slot_flag* controls when PC is loaded with *pc_temp*.

```
WHEN i_jr =>
  inst <= op_jr;
  pc_temp := reg(rs);
  delay_slot_flag := set;
```

Figure 3-20. The Jump Register Instruction

There are two *special* instructions: *syscall* and *break*. *syscall* initiates a system call trap and immediately transfers control to the exception handler. *break* initiates a

breakpoint trap and also immediately transfers control to the exception handler. Since an operating system will not be modeled, these instructions are "dummy" instructions and do not do anything useful. However, since the exception handling is modeled, *syscall* and *break* cause an exception which halts the processor.

There are eight *multiply/divide* instructions: *move from hi (mfhi), move to hi (mthi), move from lo (mflo), move to lo (mtlo), multiply (mult), multiply unsigned (multu), divide (div),* and *divide unsigned (divu).* The first four instructions move data to and from the *hi_reg* and *lo_reg* registers. The *hi_reg* and *lo_reg* registers hold results of integer multiplication and division operations. As an example, the *mult* instruction is shown in Figure 3-21. The *mult* instruction multiplies the contents of registers *rs* and *rt* as two's complement values. The *mult* function returns a value that is placed in a temporary 64-bit variable *mult_temp. mult_temp* is divided into two 32-bit values representing the most and least significant 32 bits. The most significant 32 bits are stored in *hi_reg.* The least significant 32 bits are stored in *lo_reg.*

```
WHEN i_mult =>
  inst <= op_mult;
  mult_temp := mult(reg(rs), reg(rt));
  lo_reg := mult_temp(31 DOWNTO 0);
  hi_reg := mult_temp(63 DOWNTO 32);
```

Figure 3-21. The Multiply Instruction

There are ten *3-operand register* instructions: *add (add), add unsigned (addu), subtract (sub), subtract unsigned (subu), AND (and), OR (or), XOR (xor), NOR (nor), set on less than (slt),* and *set on less than unsigned (sltu).* All ten instructions perform their operations on the 3-operand registers: *rs, rt,* and *rd.* The two operands are stored in the *rs* and *rt* registers. The results of the operation is stored in *rd.* The first four instructions are arithmetic instructions. The next four instructions perform logical operations. The

last two instructions are used to compare and set registers. As an example, the *add* instruction is shown in Figure 3-22. The *add* instruction adds the contents of registers *rs* and *rt* and places the 32-bit result in register *rd*. If an overflow occurs, the *ovrflw* bit is set to '1'. The overflow condition is checked using an IF statement. If an overflow condition exists, then *exception_flag* is set to enumerated value *overflow*.

```
WHEN i_add =>
  inst <= op_add;
  IF rd /= 0 THEN
      add_ovf(reg(rs), reg(rt), reg(rd),ovrflw);
      IF ovrflw = '1' THEN
          exception_flag := overflow;
      END IF;
  END IF;
```

Figure 3-22. The Add Instruction

BRANCH CONDITIONAL INSTRUCTIONS

There are four *branch conditional*, or *bcond*, instructions: *branch on less than zero (bltz), branch on greater than or equal to zero (bgez), branch on less than zero and link (bltzal),* and *branch on greater than or equal to zero and link (bgezal).* These instructions change the control flow of a program depending on a condition. The link instructions save a return address in register 31 (*r31*). All branch instruction target addresses are computed by adding the address of the instruction in the delay slot with the 16-bit *offset.* The 16-bit *offset* is shifted left two bits and sign extended to 32 bits. All branches occur with a delay of one instruction. As an example, the *bltz* instruction is shown in Figure 3-23. The *bltz* instruction branches to the target address if register *rs* is less than zero. The check for less than zero is accomplished by testing bit 31 of the contents of *rs*. If bit 31 is '1' then the value in *rs* is negative. The *pc_reg* variable holds

the address of present instruction, the *bltz* instruction. The address of the instruction in the delay slot is computed by adding four to *pc_reg*. The 16-bit *offset* is manipulated by first sign-extending it to 32 bits and then shifting it left by two bits. Finally, the target address is computed by adding the modified *offset* to the modified program counter. The target address is placed in a temporary program counter (*pc_temp*) since all branch instructions have a delay of one instruction. If the branch is to be taken, at the proper time, *pc_reg* is updated with *pc_temp*. *delay_slot_flag* is the variable that controls when *pc_reg* is updated.

```
WHEN i_bltz =>
  inst <= op_bltz;
  IF reg(rs)(31) = '1' THEN
      pc_temp := pc_reg + x"0000_0004";
      pc_temp := pc_temp +
                 shift_ll(se16to32(offset),2);
      delay_slot_flag := set;
  END IF;
```

Figure 3-23. The Branch on Less Than Zero Instruction

JUMP INSTRUCTIONS

There are two *jump* instructions: *jump (j)* and *jump and link (jal)*. Both instructions jump to an address contained in the *target* field. More precisely, the 26-bit *target* address is shifted left two bits and combined with the high-order 4 bits of the program counter. The program jumps to the address with a one instruction delay. The *jal* instruction also places the address of instruction following the delay slot in the link register, *r31*. As an example, the *j* instruction is shown in Figure 3-24.

```
WHEN i_j =>
  inst <= op_j;
  pc_temp := pc_reg(31 DOWNTO 28) & target &
             b"00";
  delay_slot_flag := set;
```

Figure 3-24. The Jump Instruction

BRANCH INSTRUCTIONS

There are four branch instructions: *branch on equal (beq), branch on not equal (bne), branch on less than or equal to zero (blez),* and *branch on greater than zero (bgtz).* The branch instructions are similar in operation to the branch condition instructions. The branch target address is calculated from the sum of the address of the instruction in the delay slot with the 16-bit *offset.* The 16-bit *offset* is shifted left two bits and sign-extended to 32 bits. As an example, the *beq* instruction is shown in Figure 3-25. The *beq* instruction branches to the *target* address if the contents of general register *rs* and *rt* are equal, with a delay of one instruction.

```
WHEN i_beq =>
  inst <= op_beq;
  IF reg(rs) = reg(rt) THEN
     pc_temp := pc_reg + x"0000_0004";
     pc_temp := pc_temp +
                shift_ll(se16to32(offset),2);
     delay_slot_flag := set;
  END IF;
```

Figure 3-25. The Branch on Equal Instruction

ALU IMMEDIATE INSTRUCTIONS

There are eight ALU immediate instructions: *add immediate (addi), add immediate unsigned (addiu), set on less than immediate (slti), set on less than immediate unsigned (sltiu), AND immediate (andi), OR immediate (ori), XOR immediate (xori),* and *load upper immediate (lui).* All eight instructions perform their operations using the two operand registers *rs* and *rt*, and an 16-bit immediate field. The first four instructions sign-extend the immediate field. The last four instructions zero-extend the immediate field. As an example, the *addi* instruction is shown in Figure 3-26. The *addi* instruction adds the sign-extended immediate field to the contents of general register *rs* to form a 32-bit result. This result is stored in general register *rt*. An overflow exception occurs if the two highest order carry-out bits differ. This is known as two's complement overflow.

```
WHEN i_addi =>
  inst <= op_addi;
  IF rt /= 0 THEN
      add_ovf(reg(rs), sel6to32(immed), reg(rt),
              ovrflw);
      IF ovrflw = '1' THEN
          exception_flag := overflow;
      END IF;
  END IF;
```

Figure 3-26. The Add Immediate Instruction

LOAD/STORE INSTRUCTIONS

There are seven load instructions: *load byte (lb), load halfword (lh), load word left (lwl), load word (lw), load byte unsigned (lbu), load halfword unsigned (lhu),* and *load word right (lwr).* There are five store instructions: *store byte (sb), store halfword (sh), store word left (swl), store word (sw),* and *store word right (swr).* All twelve load/store instructions calculate the effective address (*ea*) by sign-extending the 16-bit

offset and adding it to the contents of general register *base*. The contents of the memory location are sign-extended when the signed load instructions are used. When the unsigned load instructions are used, the contents are zero-extended. As an example, the *lb* instruction is shown in Figure 3-27. The *ea* is calculated by sign-extending the *offset* using the *se16to32* function and adding it to the contents of the *base* register. The memory is accessed by the *mem_read* procedure. This procedure passes the *ea* and the type of data to be read (byte) as its' arguments.

```
WHEN i_lb =>
  inst <= op_lb;
  IF rt /= 0 THEN
      ea := se16to32(offset) + reg(base);
      mem_read(ea, byte, temp_reg_val_1);
      temp_reg_num_1 := rt;
      latency_flag := set;
  END IF;
```

Figure 3-27. The Load Byte Instruction

4.0 DATAFLOW MODEL

The dataflow model is a hierarchical structure of other dataflow components. The top level of the model consists of three components: the CPU, the memory, and the compare module. These three parts comprise the dataflow test bench as shown in Figure 4-1.

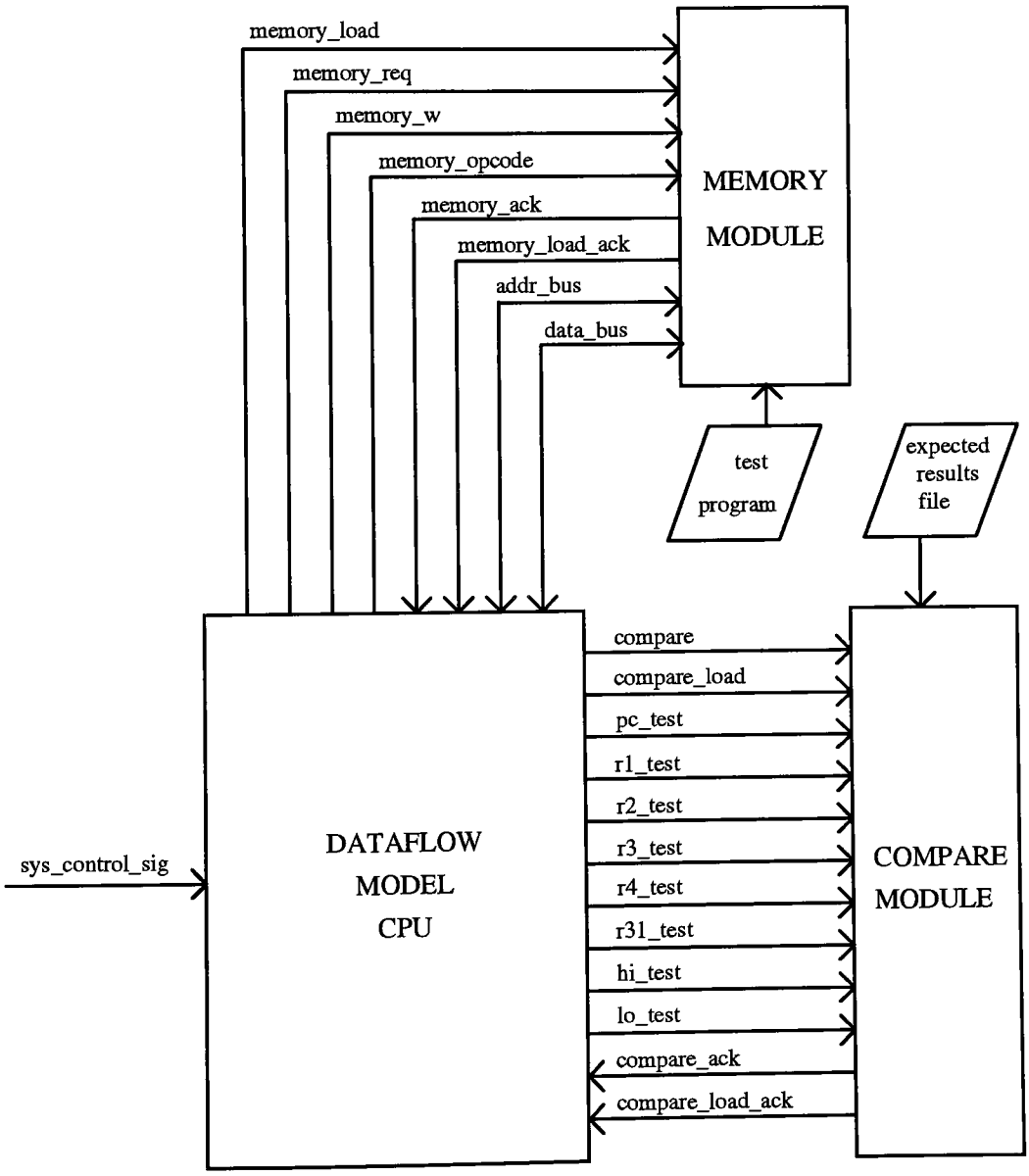


Figure 4-1. Dataflow Model Test Bench

The memory module is accessed by eight interface signals. The *memory_load* and *memory_load_ack* signals provide the handshaking necessary to load the memory with the test program. This is done when the asynchronous processor is initialized. The *memory_req* and *memory_ack* signals provide the handshaking necessary to read and write data between the CPU and memory module. The *memory_w* and *memory_opcode* signals are used in conjunction with a memory write. The last two signals, *addr_bus* and *data_bus*, correspond to the address bus and data bus, respectively. All these signals are discussed in more detail in the following sections.

The compare module is accessed by 12 interface signals. The *compare_load* and *compare_load_ack* signals provide the handshaking to load the compare module with the expected results file during system initialization. The *compare* and *compare_ack* signals provide the handshaking to test the state of the processor after each instruction. The remaining eight signals, *pc_test* through *lo_test*, monitor the contents of the specified registers.

The CPU module, shown in Figure 4-2, is composed of eight unique dataflow components. The pipeline is made up of five of these components. The pipeline stages are: instruction fetch (IF), instruction decode (ID), arithmetic logic unit (ALU), memory (MEM), and writeback (WB). Each stage of the pipeline has a handshaking control circuit (HCC) associated with it. The HCC controls the operation of the specific stage. The bus control unit (BCU) acts as a high speed polling device between the IF and MEM stages. It grants access to the address and data busses. The last component is the exception handler (EH). It sends an interrupt request and vector to the IF stage when an exception occurs from either the IF, ID, ALU, or MEM stages.

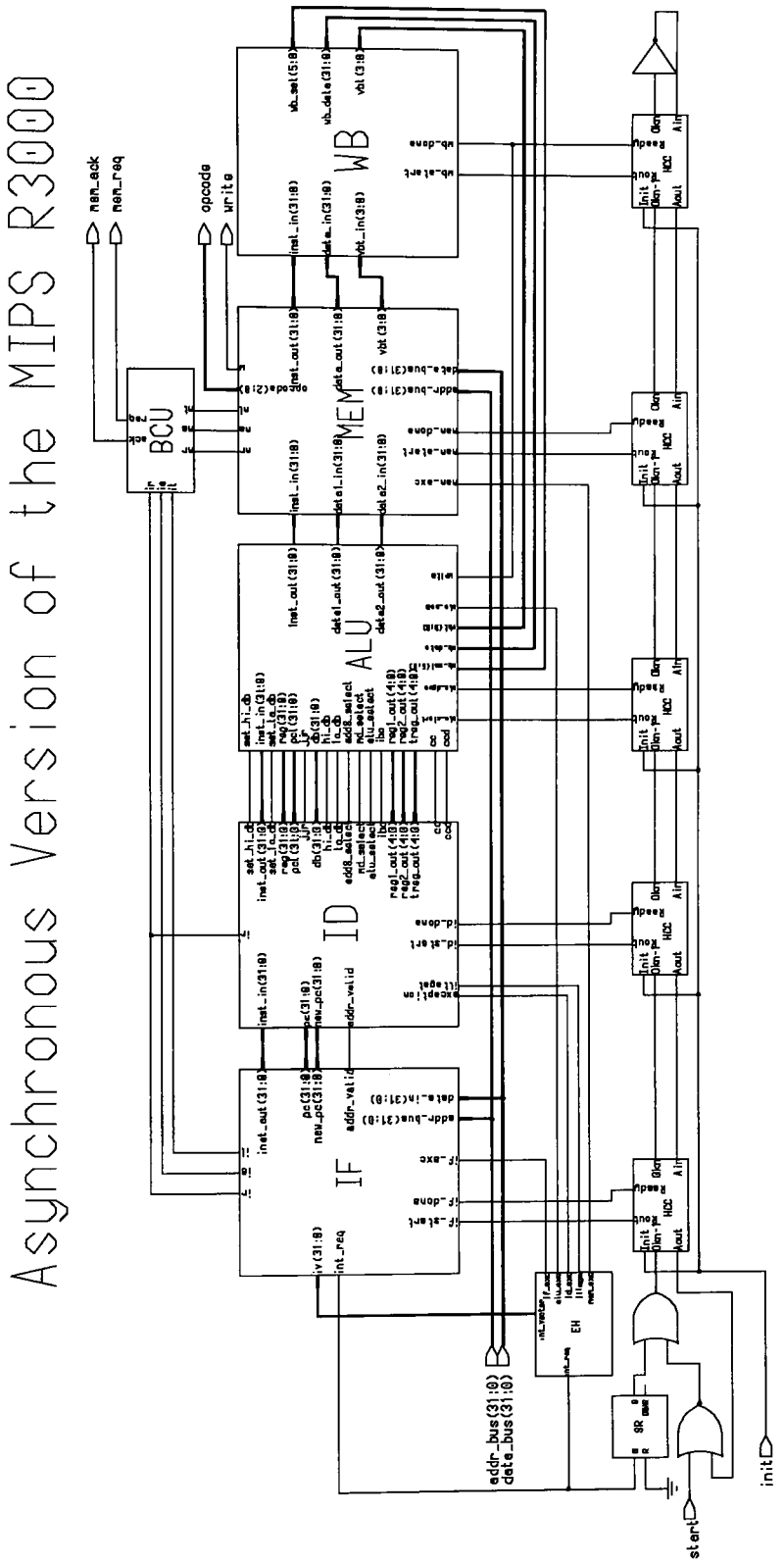


Figure 4-2. Dataflow Model CPU Component

4.1 INSTRUCTION FETCH STAGE

The IF is the first stage of the pipeline and it fetches the next instruction from memory. The first item that the IF needs is a valid address. The valid address is calculated by the address adder (AA) in the ID stage. Therefore, the IF has to wait for the AA to calculate the new address before it can use it. However, on the first instruction the IF doesn't have to wait since the program counter (PC) is initially zero. This is essential to prevent deadlock. Since the ID cannot start until the IF finishes, IF cannot initially wait for the AA which is inside the ID. Once the IF gets the valid address, it needs access to the address and data busses to retrieve the data from memory. The IF sends out a request to the BCU. The BCU grants the address bus to the IF stage if it is free. The BCU is discussed in more detail in section 4.6.

The IF also handles branching to an interrupt vector. Under normal operations, the IF just continues to fetch the next instruction once it receives a valid address. However, when an exception occurs, an interrupt and interrupt vector is generated by the EH. The IF jumps to this interrupt vector instead of the new PC value. The IF stage can also generate an exception. This occurs when the address of an instruction is not aligned on a word boundary. In other words, if the two least significant bits are not zero then an address exception is generated.

The schematic to the IF stage is shown in Figure 4-3. The IF is started by the *if_start* signal going high. This start signal is generated by the IF's HCC. Once it receives the start signal, the IF waits for either *addr_valid* to go low or *int_req* to go high. The *addr_valid* signal going low signifies that the ID has calculated the new address and this address is valid . The *int_req* signals that an interrupt request has been generated. The IF receives its new address from either the *new_pc* or *iv* lines. The *new_pc* line comes from the AA. The *iv* stands for interrupt vector and comes from the EH. These two lines are multiplexed and selected by the *int_req* line. If an interrupt occurs, then the IF uses the

interrupt vector (*iv*) value. On the other hand, if the IF is in normal operation then the new PC value (*new_pc*) is used.

When *addr_valid* goes low, the IF sends a bus request (*ir*) to the BCU. This signal is also used to latch the address that comes from the AA. The IF now waits until the BCU sets *ia* high. The *ia* signal going high is an acknowledgment from the BCU that the IF has been granted access to the address bus. When *ia* goes high, it enables the tri-state buffer, placing the address on the address bus. This *ia* acknowledgment signal is needed to allow the address to be setup on the address bus before the memory read. Now the IF sends the BCU the *il* line high signaling that the address has been loaded on the address bus. The BCU can now initiate the memory request. The IF waits until *ia* goes low signaling that the data bus now contains a valid instruction. The *ia* line going low causes the instruction on the data bus to be latched and resets the *ir* and *il* signals. Resetting the *ir* line signals to the BCU that the IF is finished with the address bus. The waveforms showing the IF operation are shown in Figure 4-4.

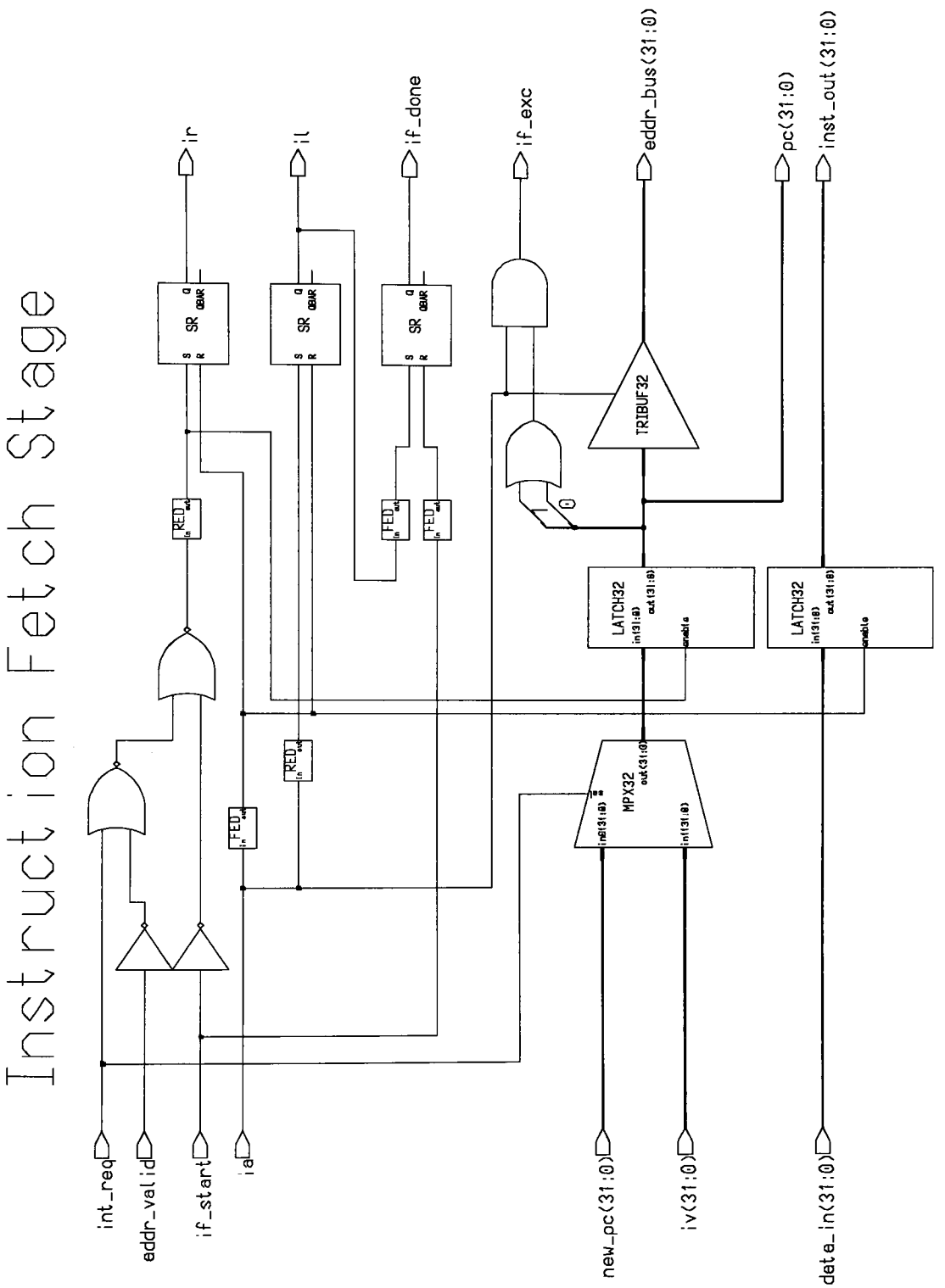


Figure 4-3. Schematic of IF Stage

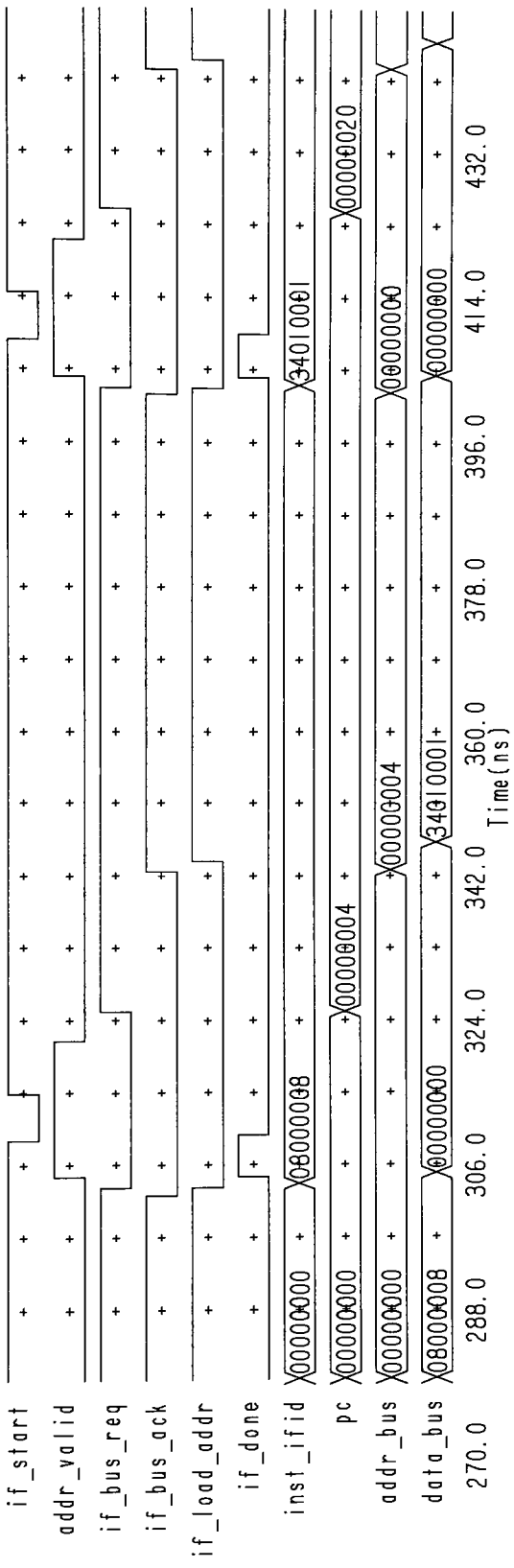


Figure 4-4. Waveforms of IF Stage

4.2 INSTRUCTION DECODE STAGE

The ID is the second stage of the pipeline and has three main tasks. The first task is to decode the instruction. The second task is to calculate the destination address for all branch and jump instructions, and to increment the PC on other instructions. The third task stalls the pipeline during data dependencies. These three tasks are explained in more detail in the following sections. The overall schematic diagram of ID is shown in Figure 4-6.

INSTRUCTION DECODER

The instruction decoder component, shown in Figure 4-5, decodes the instruction into 17 different select lines. These select lines are distributed within the ID stage and to the ALU stage. The 17 select lines are described in Table 4-1 and 4-2. A code excerpt of the instruction decoder is shown in Figure 4-7.

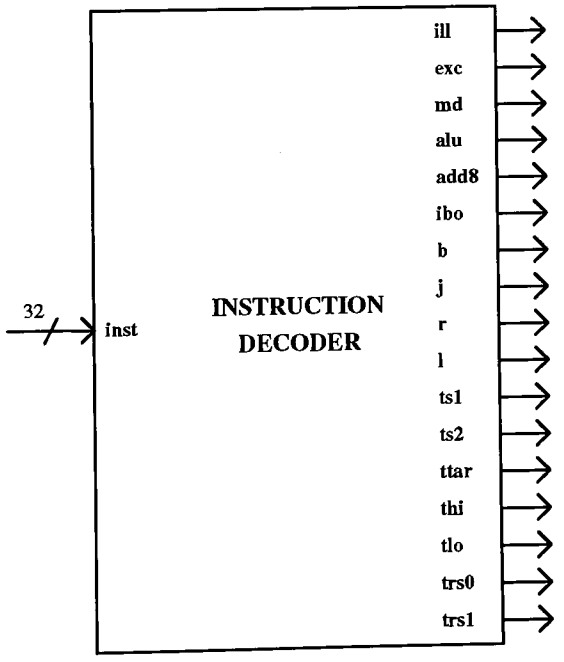


Figure 4-5. Instruction Decoder Component

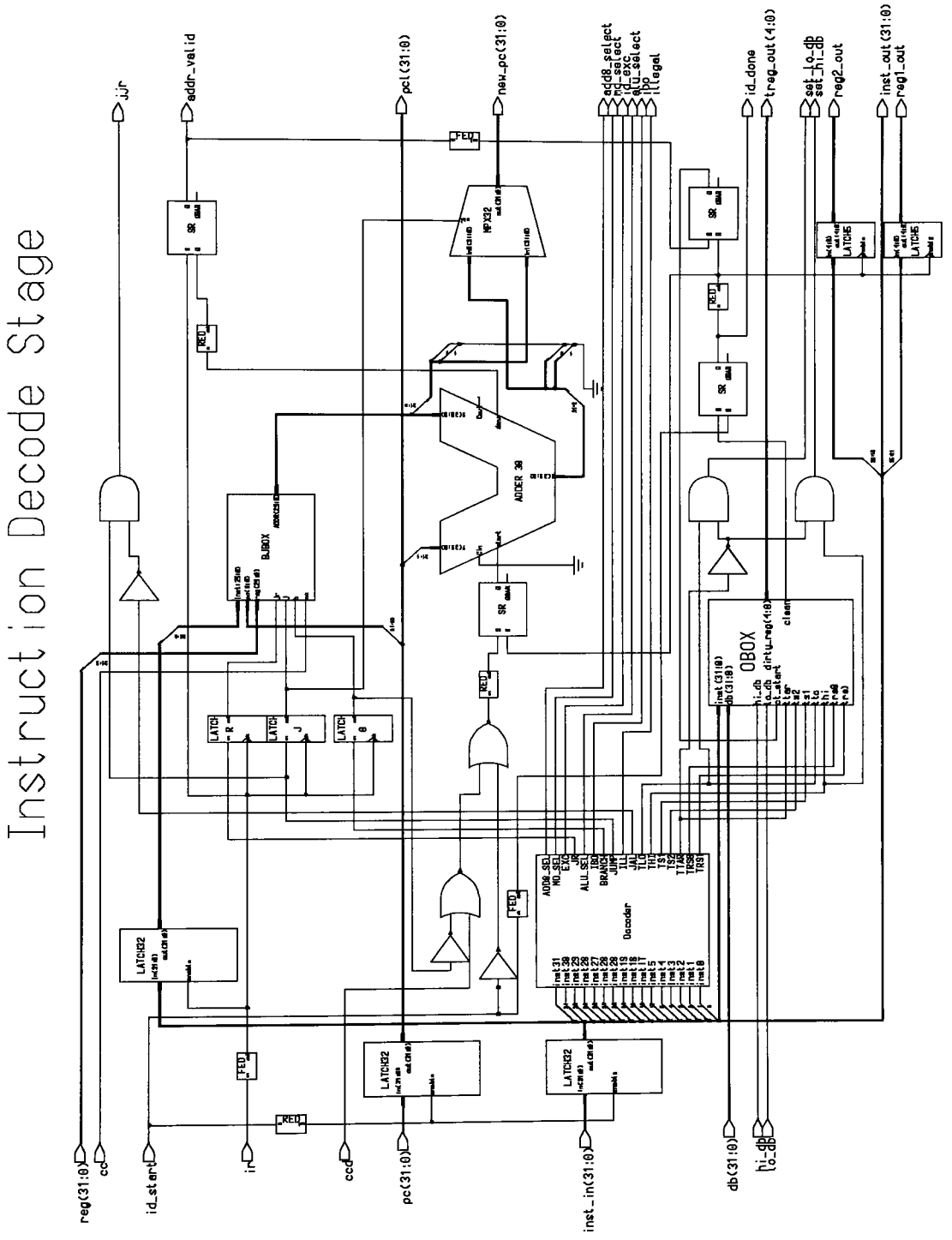


Figure 4-6. Schematic Diagram of ID Stage

SIGNAL	NAME	DESCRIPTION
ill	illegal	This signal goes high when an illegal instruction is encountered
exc	exception	This signal goes high when the instruction is *syscall* or *break*
md	MDU select	This signal goes high when an instruction needs to use the MDU in the ALU stage
alu	ALU select	This signal goes high when an instruction needs to use the ALU in the ALU stage
add8	add8 unit select	This signal goes high when an instruction needs to use the add8 unit in the ALU stage
ibo	immediate or base offset	This signal goes high either for an immediate instruction or an instruction needs a base-offset calculation
b	branch	This signal goes high on a branch instruction
j	jump	This signal goes high on a jump instruction
r	"register" instruction	This signal goes high when the instruction is a *jump register (jr)* or a *jump and link register (jalr)*
l	"link" instruction	This signal goes high when the instruction is a *jump and link (jal)* or a *jump and link register (jalr)*
ts1	test 1st source register	This signal goes high when the first source register must be tested for a data dependency
ts2	test 2nd source register	This signal goes high when the second source register must be tested for a data dependency
ttar	test target register	This signal goes high when the destination register must be tested for a data dependency
thi	test hi register	This signal goes high when the hi register must be tested for a data dependency
tlo	test lo register	This signal goes high when the lo register must be tested for a data dependency
trs0, trs1	target register select bits	These signals are used to determine the destination register, SEE TABLE 4-2 for the bit encoding

Table 4-1. Instruction Decoder Select Lines

TRS0	TRS1	5-BIT ENCODED DESTINATION REGISTER VALUE
0	0	bits 15-11 of the instruction
0	1	bits 20-16 of the instruction
1	0	set destination register to 0 ("00000") - means no destination register
1	1	set destination register to 31 ("11111") - used for link instructions

Table 4-2. Bit Encoding to Determine Destination Register

The first two signals, *ill* and *exc*, represent illegal and exception instructions, respectively. The *ill* signal is set high when the instruction decoder finds an instruction that is not part of the instruction set. The instruction set can be found in Table 3-1. The *exc* signal goes high on a *syscall* or *break* instruction. Both of these instructions cause a software exception.

The next three signals, *md, alu,* and *add8*, select the MDU, ALU, and ADD8 in the ALU stage, respectively. The MDU and ALU are never activated together by the same instruction. When the ALU is selected, the MDU is idle for the particular instruction. When the MDU is selected, the ALU is idle. Both lines are set low when an instruction does not need to use either unit. The ADD8 unit is only used by *branch conditional (bcond)* instructions which need to calculate a link address (*bltzal* and *bgezal*). The ADD8 unit is needed because the AA and the ALU are busy doing other calculations for the two *bcond* link instructions. The AA calculates the branch destination address and the ALU calculates the condition code (whether or not to take the branch).

The fifth signal, *ibo*, stands for immediate or base-offset. This signal controls a multiplexer to the ALU A-bus inside the ALU stage. It determines what value gets fed into the A input of the ALU (not the ALU stage but the ALU component). The *ibo* signal is high for an immediate or base-offset instruction. Base-offset values are calculated for all load/store and branch instructions. However, the *ibo* line is only used for load/store instructions because base-offset calculations for branch instructions are done by the AA.

The next four signals, *b, j, r,* and *l*, determine the type of jump or branch. A branch instruction is denoted when the *b* line is high. The branch target address is (PC + 4) + (offset * 4). This is calculated by adding the sum of the address of the instruction in the delay slot to a 16-bit offset which is shift left two bits and sign-extended to 32-bits. A jump instruction is denoted by the *j* line. The jump target address is PC(31:28) & target & "00". This is calculated by concatenating (&) the four most significant bits of the PC, the 26-bit *target*, and two zeros. The *r* line goes high if the jump instruction is a "register"

```
ARCHITECTURE dfinstdec_a OF dfinstdec IS
BEGIN

    md   <= '1' AFTER 4.4 ns WHEN -- a15
                                  (i(31) = '0' AND
                                   i(29 DOWNTO 26) = "0000" AND
                                   i(4 DOWNTO 3) = "11") ELSE
            '0' AFTER 4.4 ns;
.
.
.
END dfinstdec_a;
```

Figure 4-7. Code Excerpt of the ID Stage Instruction Decoder Architecture

instruction. *Jump register (jr)* and *jump and link register (jalr)* are the two "register" instructions. This *r* line is needed because a jump register instructions get their jump target address from the contents of a register. The *l* line goes high on a jump and link instruction. The two jump and link instructions are *jump and link (jal)* and *jump and link register (jalr)*. For these instructions, the ID must give the ADD8 unit in the ALU stage the PC to calculate the link address of (PC + 8).

The next five signals, *ts1, ts2, ttar, thi,* and *tlo*, are for data dependency checking. Each of these signals, when set high, tests to see if respective register is dirty. IF a register is dirty then the ID is stalled until the previous instruction writes its data to this register. Data dependency is discussed in more detail in a later section.

The last two signals, *trs0* and *trs1*, are used together to determine where in the instruction to get the destination register value. The four choices are shown in Table 4-2. When *trs0* and *trs1* equal "00" then the destination register value is found at bits 15-11 of the instruction. This choice is used for all *special* instructions. When *trs0* and *trs1* equal "01" then the destination is found at bits 20-16 of the instruction. This choice is used for loads and immediate instructions. These instructions use a 16-bit offset or immediate value located at bits 15-0 of the instruction. Therefore the destination bit location is

moved to bits 20-16. When *trs0* and *trs1* equal "11" then the destination register is set to 31 ("11111"). This is used by the jump and link (jal) instruction. All other instructions do not have a destination register. Therefore, the last choice sets the destination register to zero ("00000"). Register zero is hardwired to a value of zero.

ADDRESS ADDER

The address adder (AA) performs two functions: calculate destination addresses and increment the PC. If the instruction is a branch or a jump the AA calculates the destination address. For all other instructions, the AA increments the PC. The AA component is shown in Figure 4-8. The VHDL code is shown in Figure 4-9.

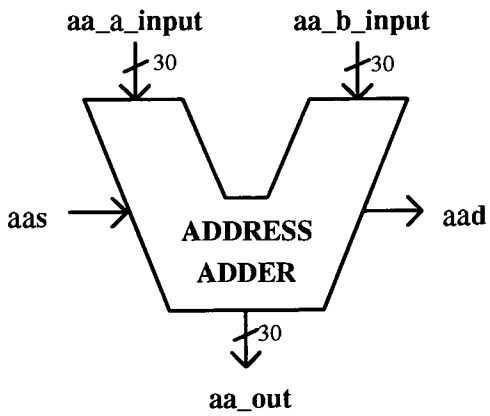


Figure 4-8. Address Adder (AA) Component

The AA has its own start and done signal. The AA start signal, *aas*, goes high when certain conditions are met: *aas* = *id_start* (NOT(*b_l*) + *ccd*). The *id_start* signal is the start signal for the ID stage. The *b_l* signal is the output of a latch that stores whether the last instruction was a branch. The condition code done (*ccd*) signal is sent by the ALU stage and indicates when the condition code evaluation (whether or not to take a branch or jump) has finished. The *aas* signal goes high when *id_start* is high and the last

instruction was not a branch. If the last instruction was a branch, then the AA has to wait until the ALU stage sends the *ccd* signal. Therefore, *aas* also goes high when *id_start* is high and *ccd* is high. The AA done signal, *aad*, is generated by the AA when it completes it operation.

```
ARCHITECTURE dfaa_a OF dfaa IS
BEGIN

    o <= a + b AFTER 7 ns WHEN s = '1' ELSE
         x"0000_0000";

    d <= '1' AFTER 8 ns WHEN s = '1' ELSE
         '0' AFTER 1 ns;

END dfaa_a;
```

Figure 4-9. Code Excerpt of the Address Adder Architecture

The AA input and output ports are all 30 bits. The two least significant bits are always zero so that the address of each instruction is aligned on a word boundary in memory. The A-input port, *aa_a_input*, gets it value from the PC. The B-input port, *aa_b_input*, gets is value from the output of the BJBOX (Branch and Jump Box) component. BJBOX selects what gets added to the PC (A-input) depending on the type of the previous instruction. The BJBOX component is shown in Figure 4-10. The schematic diagram is shown in Figure 4-12. A code excerpt is shown in Figure 4-11.

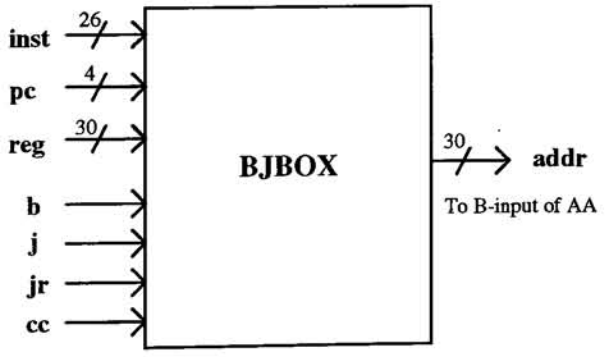


Figure 4-10. Branch and Jump (BJBOX) Component

The *inst* signal provides the lower 26 bits of the instruction. A *jump* instruction uses all 26 bits and is called the *target* value. A *branch* instruction uses only the lower 16 bits and is called an *offset* value. When the condition code , *cc*, is high then the branch is taken. The *pc* signal provides the upper four bits of the PC. This is needed to calculate the jump target address. The *reg* signal is used with the *register jump* instructions. This signal is the register jump target address. The *b, j,* and *jr* signals correspond to a *branch, jump,* and *register jump* instructions, respectively. The *addr* signal is the BJBOX output and is the address that is passed to the AA.

```
ARCHITECTURE dfbjbox_a OF dfbjbox IS

    -- component and signal declarations

BEGIN

    seb <= inst26(15);
    seb4 <= seb & seb & seb & seb;
    seb10 <= seb & seb & seb4 & seb4;

    mux1: df4to1mux4
        PORT MAP(z4, reg(29 DOWNTO 26), pc4, seb4, s0, s1, m1);

    mux2: df4to1mux10
        PORT MAP(z10, reg(25 DOWNTO 16), inst26(25 DOWNTO 16), seb10,
                 s0, s1, m2);

    mux3: df4to1mux16
        PORT MAP(iv, reg(15 DOWNTO 0), inst26(15 DOWNTO 0),
                 inst26(15 DOWNTO 0), s0, s1, m3);

    addr30 <= m1 & m2 & m3;

    s0 <= NOT (x AND y) AFTER 0.7 ns;
    s1 <= NOT (x AND z) AFTER 0.7 ns;

    x  <= NOT (cc AND b) AFTER 0.7 ns;
    y  <= NOT jr AFTER 0.3 ns;
    z  <= NOT (y AND j) AFTER 0.7 ns;

END dfbjbox_a;
```

Figure 4-11. Code Excerpt of the BJBOX Architecture

BJBOX has to handle four different instruction cases: *branches, jumps, register jumps,* and all other instructions. For the branch instruction case, BJBOX sign-extends

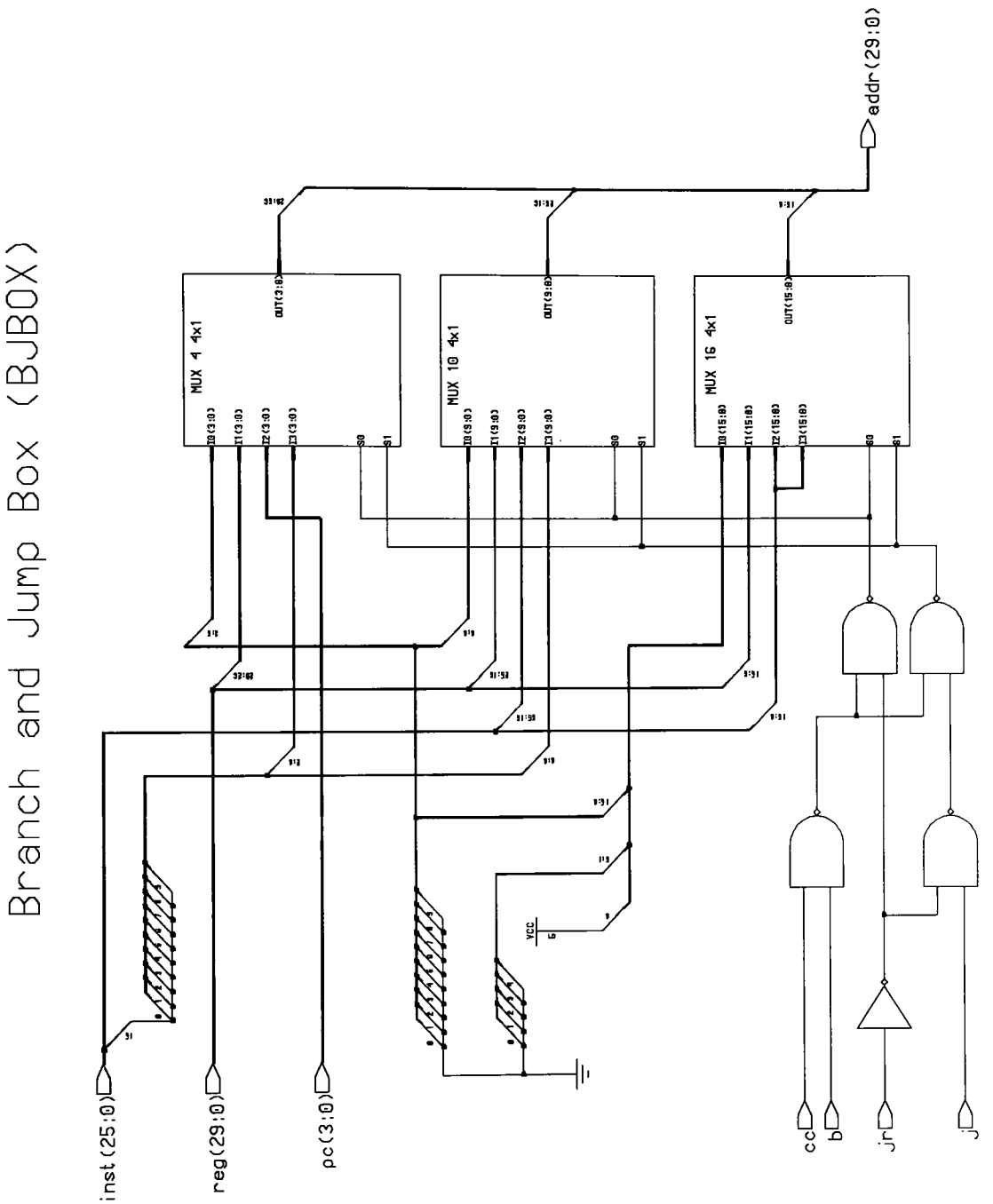


Figure 4-12. Schematic Diagram of BJBOX

the 16-bit *offset* of the *branch* instruction and passes it to the AA. For the jump instruction case (*j* and *jal*), the *pc*, *target*, and two zeros ("00") are concatenated together to form the jump target address. For the register jump instruction case (*jr* and *jalr*), the value in the *reg* signal is loaded into the PC. The last case is used for all other instructions. Here, a value of four (x"0000_0004") is passed to the AA. This allows the PC to be incremented by one word.

DATA DEPENDENCIES

A data dependency occurs when two adjacent instructions try to share the same resources. As an example, consider the two instructions shown in Figure 4-13. The first instruction places its results in register three (*r3*) when it reaches the WB stage. However, before instruction 1 can write back its answer, the second instruction tries to use it. If this data dependency problem is left unchecked, instruction 2 would receive an invalid value for *r3*. For proper operation, the second instruction has to wait until the first instruction is finished and writes back its results. Only then can instruction 2 use the value in *r3*.

```
(1)     add r3 r1 r2
(2)     add r5 r3 r4
```

Figure 4-13. Data Dependency Example

The data dependency problem is handled by tagging a register when it is "dirty" (i.e. specified as the destination by a previous instruction). This is done by using extra bits for each register, called dirty bits. Two bits are used to avoid a mutual exclusion problem. This problem exists because two different stages of the pipeline have to access these dirty bits. The ID has to set the dirty bits when a register is used as a destination. The WB has

to reset the dirty bits when it writes the data back to the register. Both stages could access the dirty bits simultaneously leading to unexpected results. The mutual exclusion problem is avoided by exclusive ORing the two bits together producing one "dirty bit". The two bits are named *db_id* and *db_wb*, and are set by the ID and WB stages, respectively. The operation of the "dirty bits" are as follows. When both bits are either zero ("00") or one ("11") the register is clean. However, if the bits differ ("01" or "10") then the register is dirty. For more information on data dependencies and the dirty bits, see "Design and Implementation of an Asynchronous Version of the MIPS R3000 Microprocessor" by Kevin Johnson [10].

The unit that handles data dependencies is called Dirty Box (DBOX) and is shown in Figure 4-14. DBOX has two tasks. The first task is to set the destination register of the current instruction as dirty (if there is a destination register). The second task is to test the source and destination registers of the current instruction to see if they were set dirty by the previous instruction.

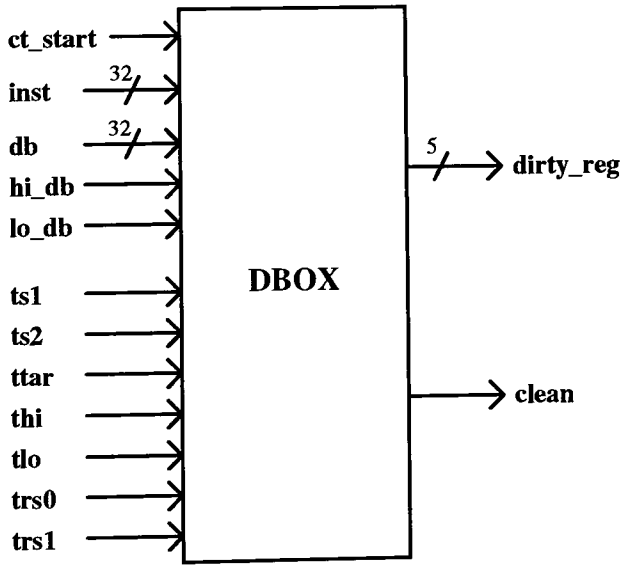


Figure 4-14. The Dirty Box (DBOX) Component

The first task is implemented by a unit inside DBOX called Target Register Dirty Select (TRDS), shown in Figure 4-15. TRDS selects which register will be set dirty. TRDS input ports are *inst, trs0,* and *trs1*. The *inst* signal is needed to determine which register is the destination. The *trs0* and *trs1* signals specify where to find the destination register. These two signals are discussed in instruction decoder section and can be found in Tables 4-1 and 4-2. The schematic diagram of TRDS is shown in Figure 4-18. The VHDL code of the TRDS architecture is shown in Figure 4-16.

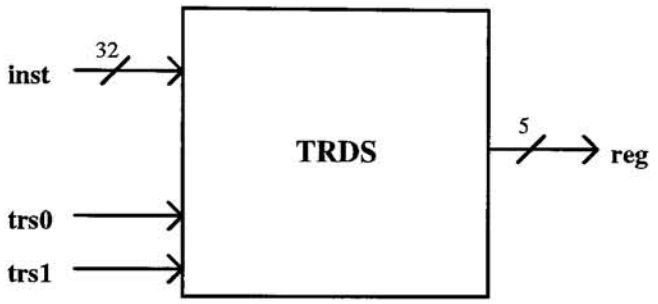


Figure 4-15. Target Register Dirty Select (TRDS) Component

```
ARCHITECTURE dftrds_a OF dftrds IS
BEGIN
    reg <= inst(15 DOWNTO 11) AFTER 0.4 ns WHEN          -- special
                        trs0 = '0' AND trs1 = '0' ELSE
           inst(20 DOWNTO 16) AFTER 0.4 ns WHEN          -- imm or load
                        trs0 = '0' AND trs1 = '1' ELSE
           b"11111" AFTER 0.4 ns WHEN                    -- jal
                        trs0 = '1' AND trs1 = '1' ELSE
           b"00000" AFTER 0.4 ns;
END dftrds_a;
```

Figure 4-16. VHDL Code of the TRDS Architecture

The second task is handled by DBOX and the schematic diagram is shown in Figure 4-19. The dirty bit signals, *db, hi_db,* and *lo_db*, come from the register banks in the ALU stage and are used for dirty bit testing. The *db* signal is a 32-bit line, each line

representing one dirty bit of the 32 general purpose registers. The *hi_db* and *lo_db* signals represent the *hi* and *lo* register dirty bits, respectively. The *ct_start* line starts the dirty bit testing. When this line is low, the dirty bit testing is valid. The five signals, *ts1, ts2, ttar, thi,* and *tlo*, determine which registers to test. These signals are generated by the instruction decoder and are discussed in the instruction decoder section. Tables 4-1 and 4-2 give a brief description of each signal. The *dirty_reg* signal is five bits wide and is the TRDS output. It specifies which destination register will be set dirty. If there is no destination register for the current instruction, then register zero is specified. The last signal, *clean*, stalls the ID stage when there is a data dependency. The VHDL code of the DBOX architecture is shown in Figure 4-17.

```
ARCHITECTURE dfdbox_a OF dfdbox IS

    COMPONENT dftrds
        PORT(inst: IN  bit_32;
             trs0: IN  bit;
             trs1: IN  bit;
             reg:  OUT bit_5);
    END COMPONENT;

    COMPONENT df32to1mux
        port(i: IN bit_32;
             s: IN bit_5;
             o: OUT bit);
    END COMPONENT;

    SIGNAL a: BIT := '1';
    SIGNAL b, d, e, f, g, h: BIT;
    SIGNAL n, p, q: BIT;
    SIGNAL reg: bit_5;
    SIGNAL c: bit;

BEGIN

    clean <= NOT (a OR b OR c) AFTER 1.1 ns;
    dirty_reg <= reg;

    c <= NOT ct_start AFTER 0.3 ns;

    a <= NOT (d AND e) AFTER 0.7 ns;
    b <= NOT (f AND g AND h) AFTER 0.8 ns;
    d <= NOT (hi_db AND thi) AFTER 0.7 ns;
    e <= NOT (lo_db AND tlo) AFTER 0.7 ns;
    f <= NOT (ts1 AND n) AFTER 0.7 ns;
    g <= NOT (ts2 AND p) AFTER 0.7 ns;
    h <= NOT (ttar AND q) AFTER 0.7 ns;

    mux1: df32to1mux
        PORT MAP (db, inst(25 DOWNTO 21), n);

    mux2: df32to1mux
        PORT MAP (db, inst(20 DOWNTO 16), p);

    mux3: df32to1mux
        PORT MAP (db, reg, q);

    trds: dftrds
        PORT MAP (inst, trs0, trs1, reg);

END dfdbox_a;
```

Figure 4-17. VHDL Code of the DBOX Architecture

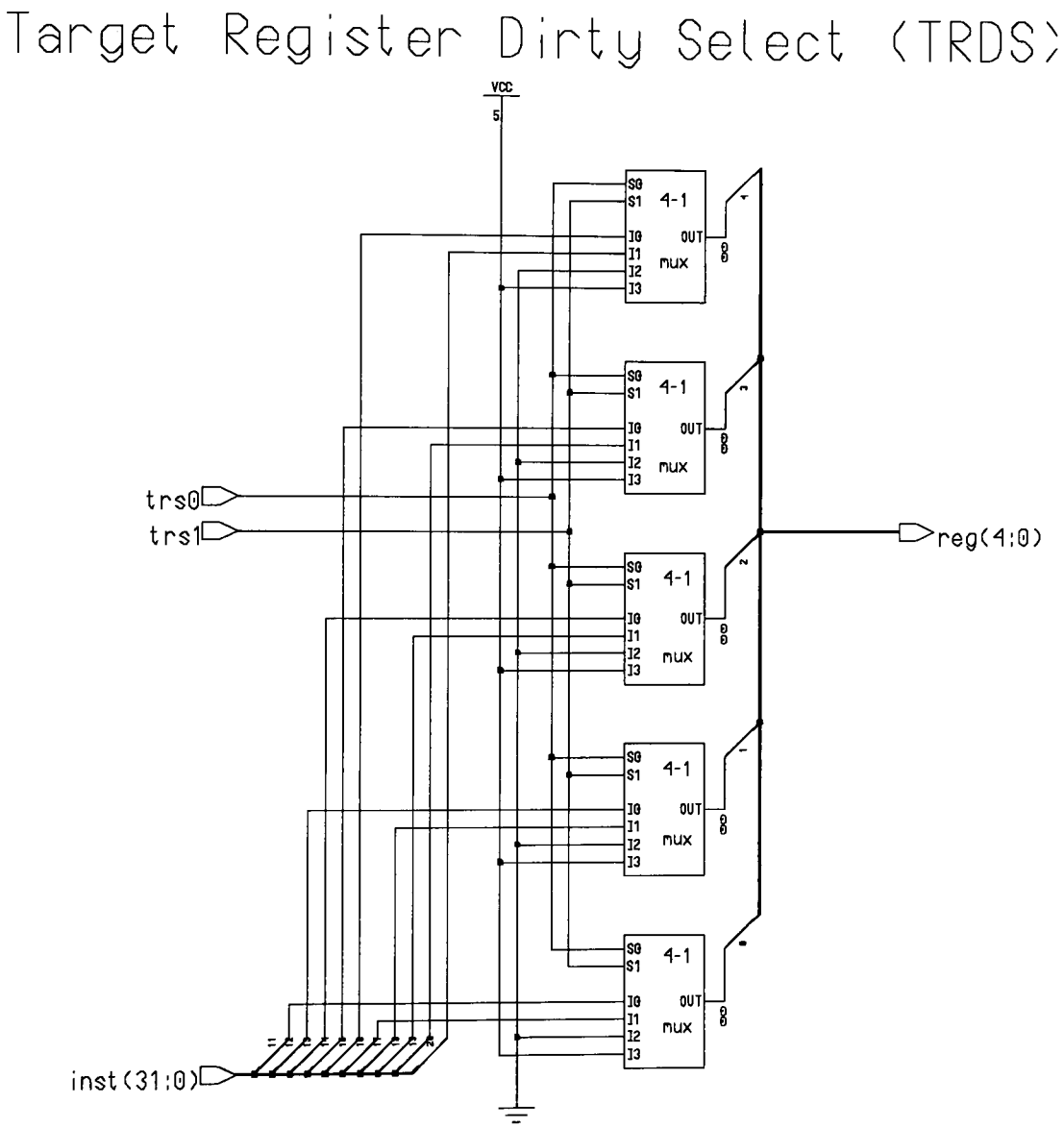


Figure 4-18. Schematic Diagram of TRDS

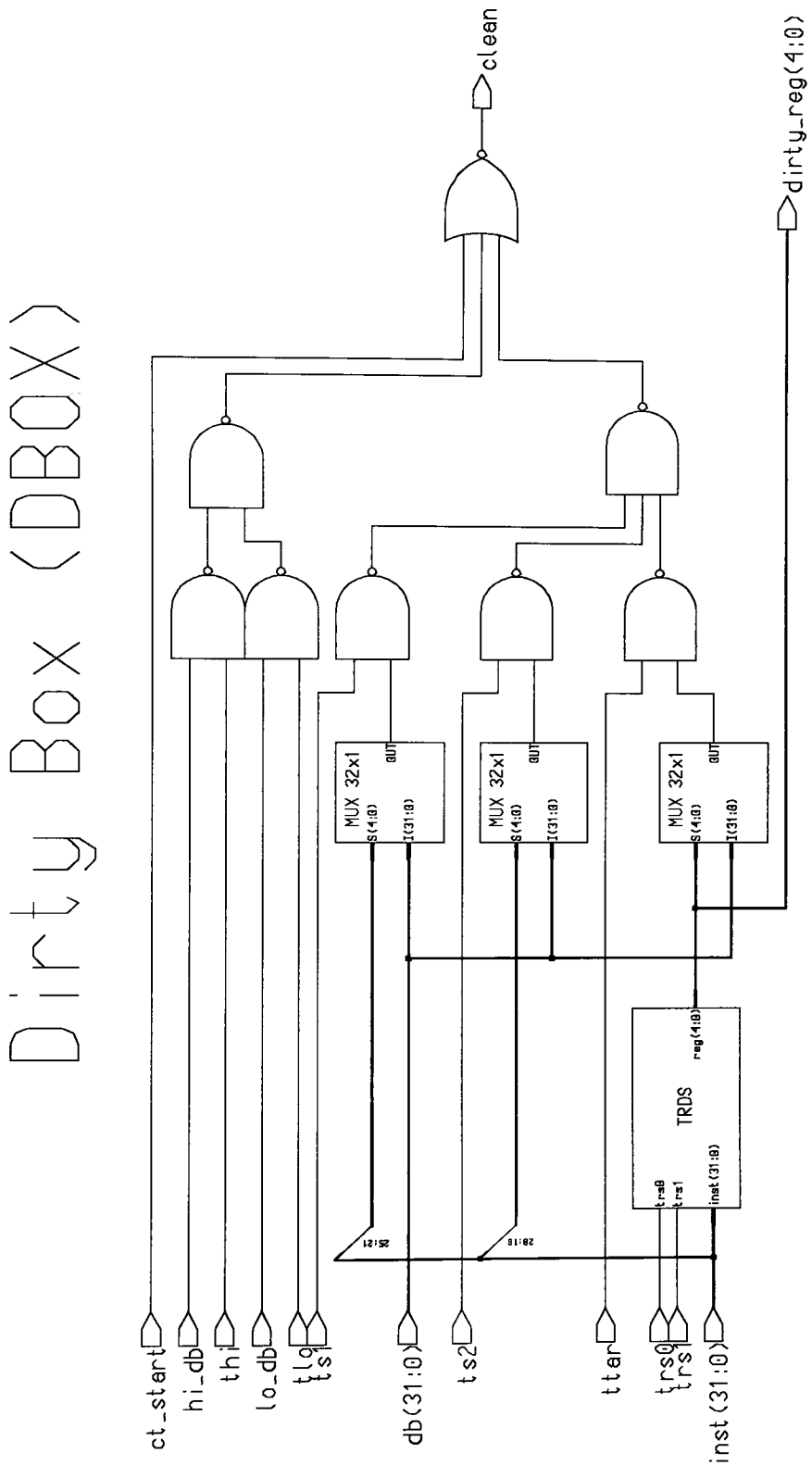


Figure 4-19. Schematic Diagram of DBOX

4.3 ARITHMETIC LOGIC UNIT STAGE

The ALU is the third stage of the pipeline and has three units running in parallel. They are the Arithmetic Logic Unit Block (ALUB), the Multiplier/Divider Unit (MDU), and the ADD8 unit. The ALU stage is responsible for all arithmetic calculations. The ALUB handles addition, subtraction, shifting, comparing, and logical operations. The MDU handles multiplication and division operations. The ADD8 unit is responsible for calculating link addresses for *branch conditional* instructions. The ALU stage also contains the general register bank and the hi/lo register bank. The ALU diagram is shown in Figure 4-20.

The MDU and ADD8 were designed by Scott Siers and are discussed in "Design and Implementation of an Asynchronous Version of the MIPS R3000 Microprocessor" [19]. The ALUB and register banks are discussed in section 5.

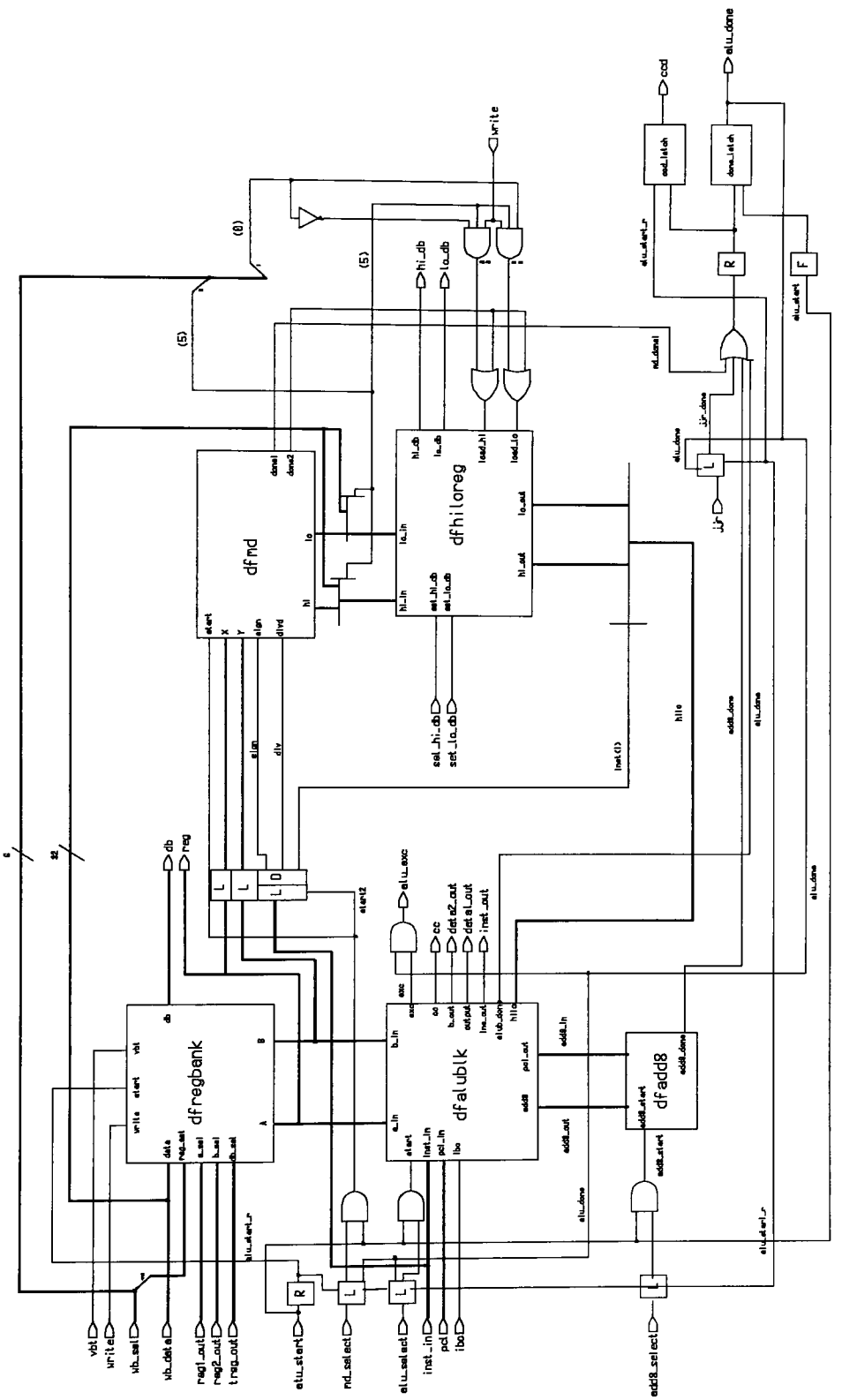


Figure 4-20. ALU Stage Block Diagram

4.4 MEMORY STAGE

The MEM is the fourth stage of the pipeline and handles all accesses to memory. This stage is very similar to the IF stage with the following exceptions. The MEM stage only accesses memory on load and store instructions while the IF does so for every instruction. MEM accesses memory by reading and writing while IF only reads. The final difference is that MEM needs to filter data read from memory. The schematic drawing is shown in Figure 4-21.

When MEM encounters a non load or store instruction, it passes the instruction on to the next stage. However, when the instruction is a load or store, the MEM stage has to gain access to the busses to carry out the instruction. MEM sends a bus request to the BCU. Once the address and data busses are free, The BCU grants them to MEM. The BCU is discussed in more detail in section 4.6.

Part of the MEM-to-memory interface involves the *write* and *opcode* signals. The *write* signal specifies whether the memory access will be a read or write. When *write* is high the access is a write. MEM has a *write* signal because it is the only stage that writes to memory (the IF stage only has to read instructions from memory). The *opcode* signal specifies the type of store instruction. The five choices are *store byte (sb), store halfword (sh), store word left (swl), store word (sw),* and *store word right (swr).* The *write* and *opcode* signals are set to default values to allow proper operation of the IF stage. The *write* signal is set to '0' to specify a read operation. The *opcode* signal is set to "011" to specify a *word.*

The MEM stage consists of three main units. They are the memory decoder, the mask unit (MU), and the shift unit (SU). The memory decoder provides necessary instruction decoding used by MEM and WB. The decoder select lines include *vbt, load, store, sus0-3,* and *mus0-3.* The valid byte tag (*vbt*) signal is used by the register bank in the ALU stage. It specifies which bytes of data are valid. Only a valid byte is written

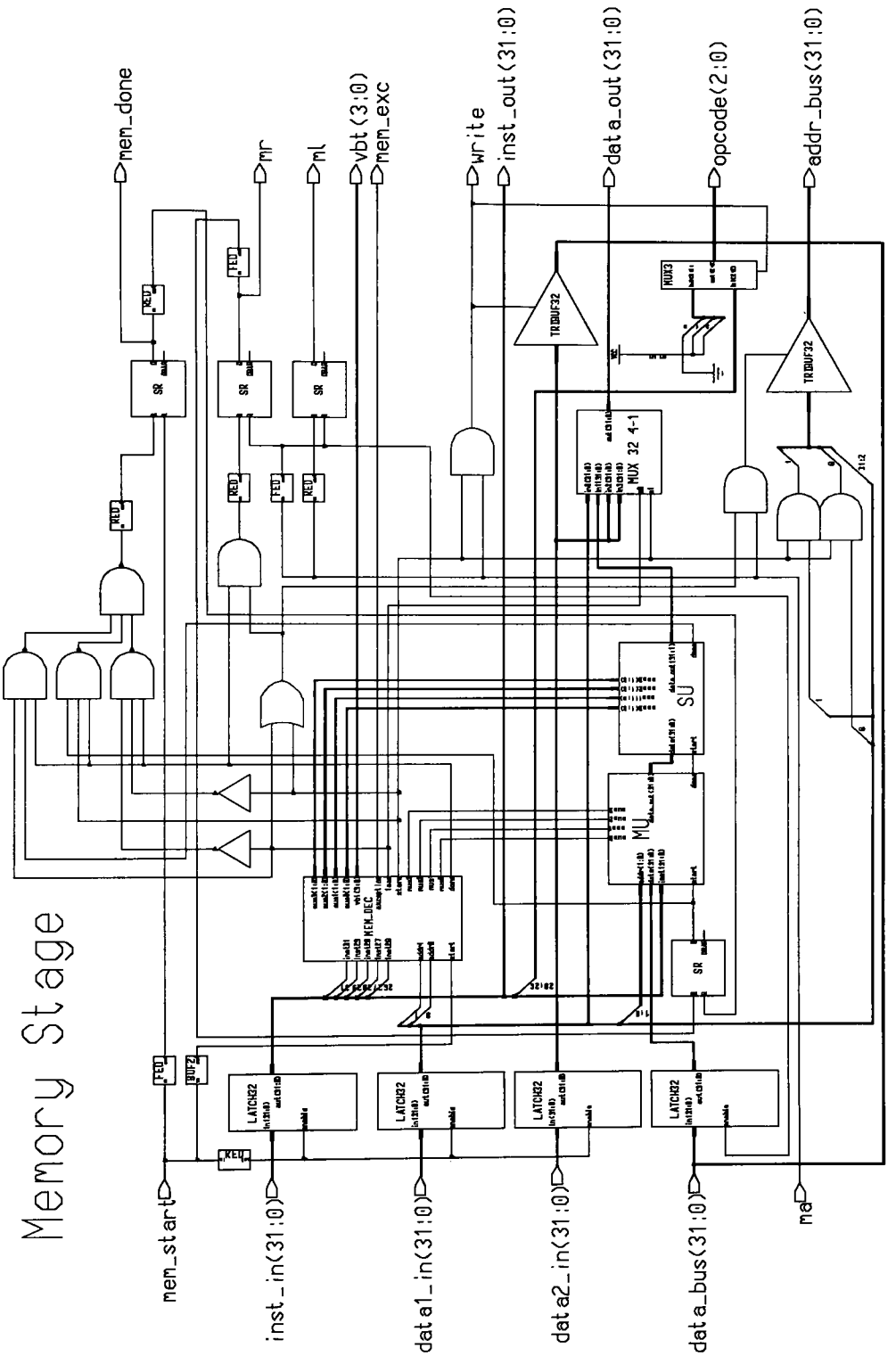


Figure 4-21. Schematic Diagram of MEM Stage

back to a register. The *load* and *store* signals specify whether the instruction is a load, store, or neither. The shift unit select (*sus0-3*) and mask unit select (*mus0-3*) signals go to the SU and MU, respectively. The MU determines the length of the requested data, and whether the data is sign-extended or zero-extended. The SU shifts data when it is not aligned on a word boundary. The VHDL code excerpts of the MU and SU are shown in Figures 4-22 and 4-23, respectively. For a more complete discussion, see "Design and Implementation of an Asynchronous Version of the MIPS R3000 Microprocessor" by Scott Siers [19].

```
ARCHITECTURE dfmu_a OF dfmu IS

    -- component and signal declarations

BEGIN

    done <= '1' AFTER 3.4 ns WHEN start = '1' ELSE
            '0' AFTER 1 ns;

    mux0: df2to1mux8 PORT MAP
        (se_byte, data(7 DOWNTO 0), mus0, data_out(7 DOWNTO 0));

    mux1: df2to1mux8 PORT MAP
        (se_byte, data(15 DOWNTO 8), mus1, data_out(15 DOWNTO 8));

    mux2: df2to1mux8 PORT MAP
        (se_byte, data(23 DOWNTO 16), mus2, data_out(23 DOWNTO 16));

    mux3: df2to1mux8 PORT MAP
        (se_byte, data(31 DOWNTO 24), mus3, data_out(31 DOWNTO 24));

    se_byte <= se_nibble & se_nibble;
    se_nibble <= seb & seb & seb & seb;

    seb  <= '1'   AFTER 2.1 ns WHEN -- a1
            (inst(28) = '0' AND data(15) = '1' AND addr = "10") OR
                                    -- a2
            (inst(28) = '0' AND data(31) = '1' AND addr = "00") OR
                                    -- a3
            (inst(28) = '0' AND data(7) = '1' AND addr = "11") OR
                                    -- a4
            (inst(28) = '0' AND data(23) = '1' AND addr = "01") ELSE
            '0'   AFTER 2.1 ns;

END dfmu_a;
```

Figure 4-22. Code Excerpt of the MU Architecture

```
ARCHITECTURE dfsu_a OF dfsu IS

    COMPONENT df4to1mux8
        PORT (i0: IN  bit_8;
              i1: IN  bit_8;
              i2: IN  bit_8;
              i3: IN  bit_8;
              s0: IN  bit;
              s1: IN  bit;
              o:  OUT bit_8);
    END COMPONENT;

BEGIN

    done <= '1' AFTER 1.4 ns WHEN start = '1' ELSE
            '0' AFTER 1 ns;

    mux0: df4to1mux8
        PORT MAP(mu_data(7 DOWNTO 0), mu_data(15 DOWNTO 8),
                 mu_data(23 DOWNTO 16), mu_data(31 DOWNTO 24),
                 sus0(0), sus0(1), data_out(7 DOWNTO 0));

    mux1: df4to1mux8
        PORT MAP(mu_data(7 DOWNTO 0), mu_data(15 DOWNTO 8),
                 mu_data(23 DOWNTO 16), mu_data(31 DOWNTO 24),
                 sus1(0), sus1(1), data_out(15 DOWNTO 8));

    mux2: df4to1mux8
        PORT MAP(mu_data(7 DOWNTO 0), mu_data(15 DOWNTO 8),
                 mu_data(23 DOWNTO 16), mu_data(31 DOWNTO 24),
                 sus2(0), sus2(1), data_out(23 DOWNTO 16));

    mux3: df4to1mux8
        PORT MAP(mu_data(7 DOWNTO 0), mu_data(15 DOWNTO 8),
                 mu_data(23 DOWNTO 16), mu_data(31 DOWNTO 24),
                 sus3(0), sus3(1), data_out(31 DOWNTO 24));

END dfsu_a;
```

Figure 4-23. Code Excerpt of the SU Architecture

4.5 WRITEBACK STAGE

The WB is the fifth and final stage of the pipeline. It writes the results back into the general purpose register bank or into the *hi/lo* register bank, depending on the instruction. The schematic diagram is shown in Figure 4-24. WB consists mainly of the decoder unit, another TRDS unit, and an encoder. The decoder unit decodes the *trs0* and *trs1* select lines used by the TRDS unit. The TRDS unit specifies which of the general purpose registers is the destination (if there is one). A six-bit encoding scheme, shown in Table 4-3, is used to determine the destination register. If the most significant bit (MSB) is '0', the least significant five bits selects one of the general purpose registers as the destination. If the MSB is '1', then the least significant bit (LSB) selects the *hi/lo* register. If the LSB is '0' then the destination is the *hi* register. The LSB equal to '1' specifies the *lo* register.

BITS 5-0	DESTINATION REGISTER
0rrrrr	general purpose register "rrrrr"
1xxxx0	*hi* register
1xxxx1	*lo* register

note: "rrrrr" = 5 bits to select which GPR
'x' = don't care

Table 4-3. WB Stage Destination Register Bit Encoding Scheme

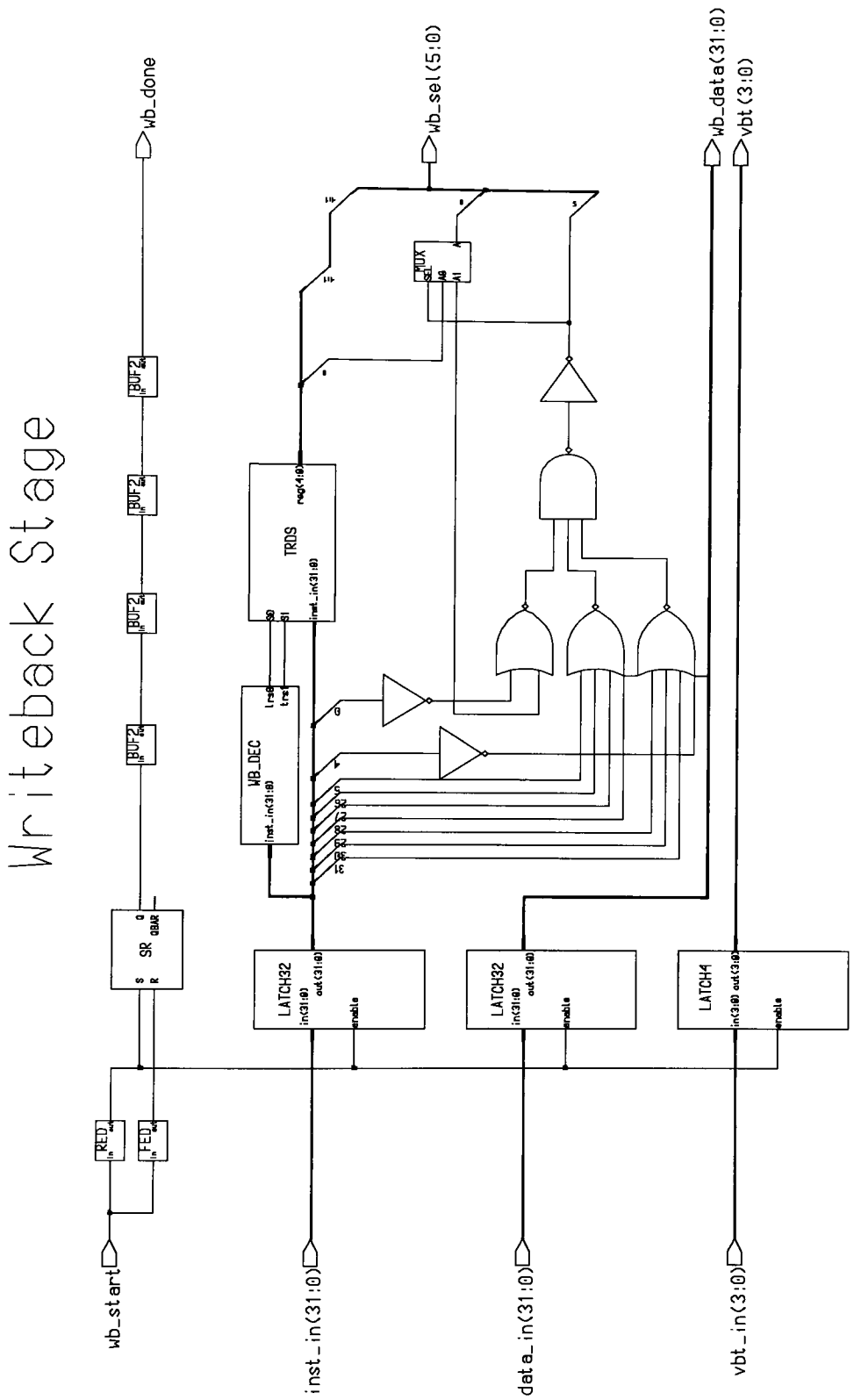


Figure 4-24. Schematic Diagram of WB Stage

4.6 BUS CONTROL UNIT

The BCU controls access to the data and address busses. It acts as an arbiter between the IF and MEM stages. When the BCU receives bus requests from both, it grants access to one stage and blocks the other. The stage that was denied access has to wait until the stage that has access is done using the busses. The interface between the BCU and each stage consists of three signals: bus request (*ir* or *mr*), bus acknowledgment (*ia* or *ma*), and address load (*il* or *ml*). The signal names that start with an "i" indicate the IF stage. Those with an "m" indicate the MEM stage. The BCU also has to interface with the memory. This interface consists of a memory request (*req*) and memory acknowledgment (*ack*). These interfaces are shown in Figure 4-25.

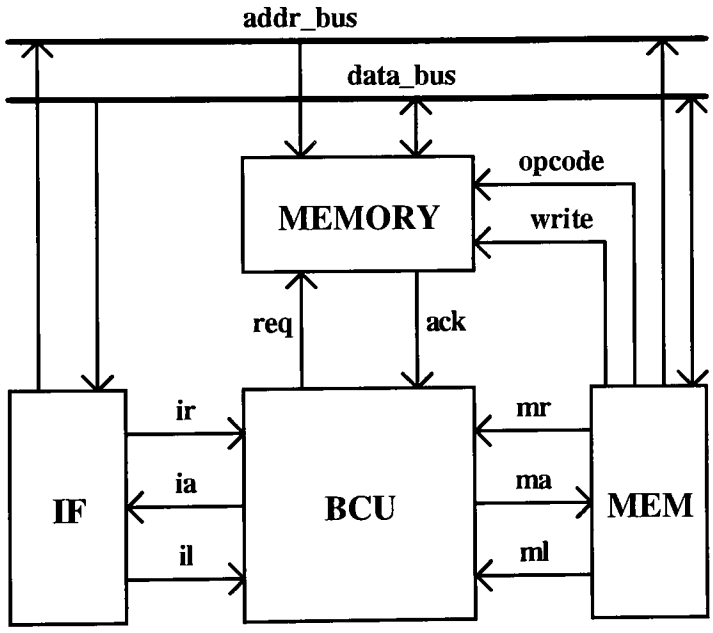


Figure 4-25. BCU Interfaces to IF, MEM, and Memory

The BCU schematic diagram is shown in Figure 4-26. The circuit operation is as follows. A stage requests memory by setting its request signal high (*ir* for IF and *mr* for MEM). The BCU selects the stage that will get the busses by setting the appropriate

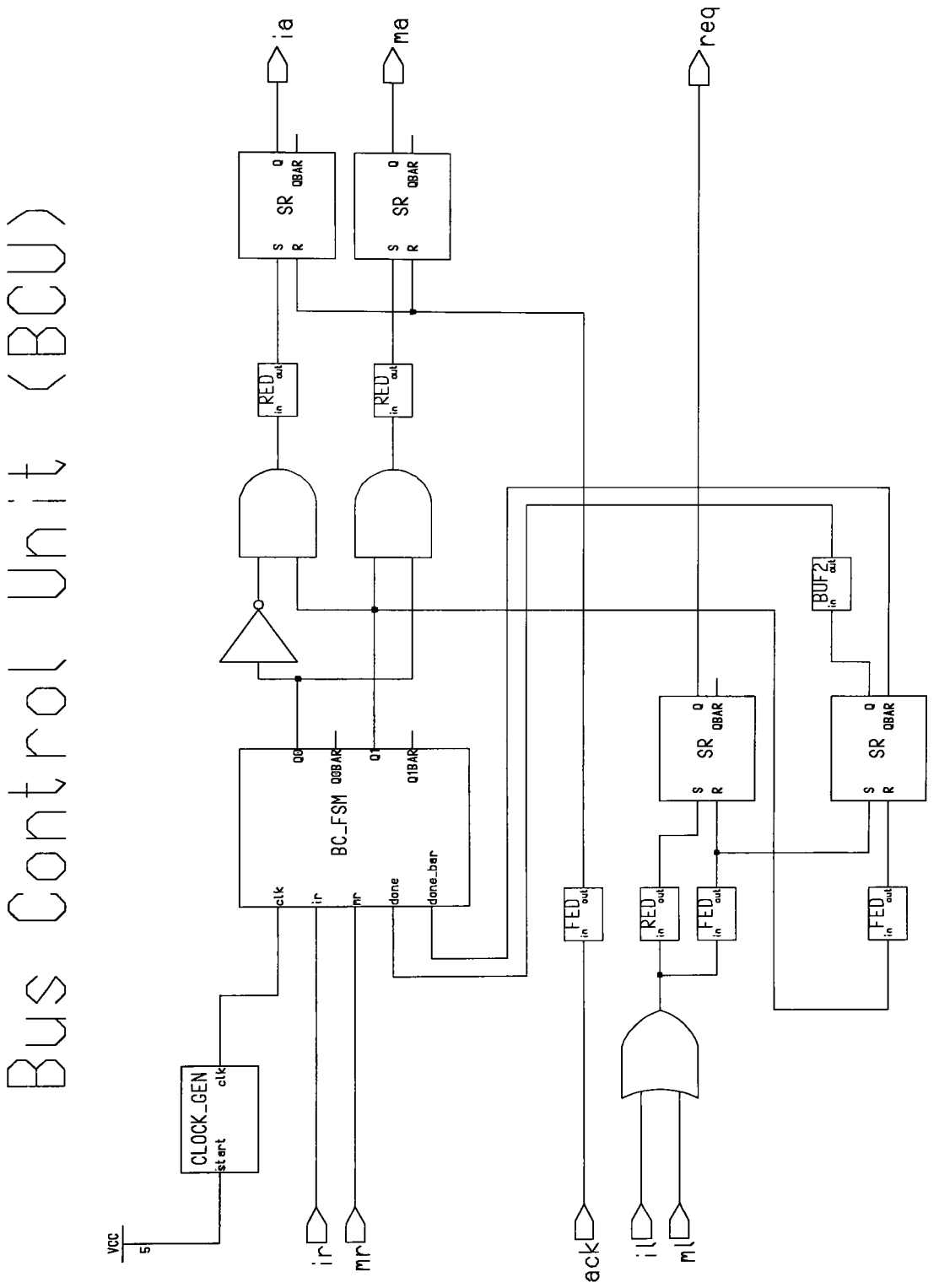


Figure 4-26. Schematic Diagram of Bus Control Unit (BCU)

acknowledgment signal (*ia* for IF and *ma* for MEM). This decision is made by polling the request lines at a high speed. Once a stage sees that it's acknowledgment line is high, it loads the appropriate values on the busses. Only the address is loaded for the IF stage and a MEM read operation Both the address and data busses are loaded for a MEM write operation. Once the busses are loaded, the stage sets it's load signal (*il* for IF and *ml* for MEM). This alerts the BCU that the busses are setup. The BCU can now perform the specified memory operation. The BCU sets the *req* signal high to send a memory request. The memory responds by setting the acknowledgment (*ack*) signal high. When memory sends the *ack* signal low, the memory operation is complete. The BCU can now reset the *req* line back to low. When the memory operation is complete, the BCU sends the appropriate acknowledgment signal low. This signals to the stage that the operation is over. The stage now resets its' request and load lines. The BCU waveforms are shown in Figure 4-27.

The *write* and *opcode* signals are only used by MEM. The IF does not need these signals because it only performs a load word operation. Therefore, MEM has to set these signals to their default settings for proper IF operation. The *write* signal specifies whether the memory operation is a read or a write. The default setting is a '0' which is a read operation. The *opcode* signal specifies the type of operation (*byte, halfword,* or *word*). The default setting is the type *word.*

The BCU is implemented using a finite state machine (FSM). The FSM has four states and is shown in Figure 4-28. State zero represents the BCU polling the IF stage. The IF is granted access to the busses if the IF's bus request line (*ir*) goes high during state zero. State one represents the BCU polling the MEM stage. The MEM is granted access if the *mr* signal goes high during state one. BCU continues to alternate between state zero and one at a high speed. This is called high speed polling. The other two states represent a stage having control of the bus. State two represents the IF and state three

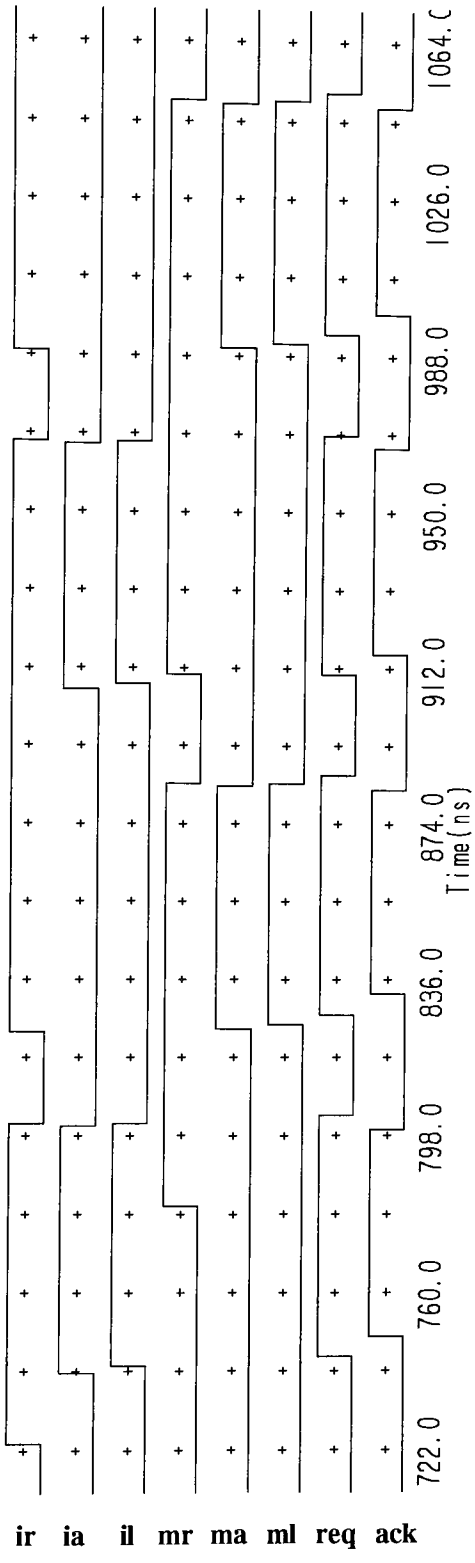
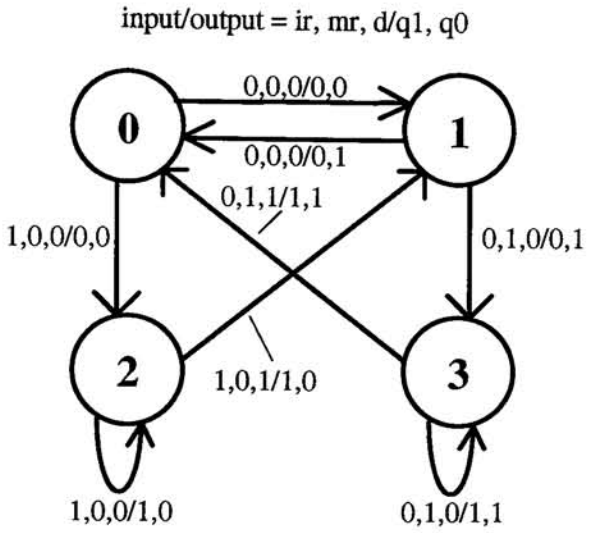


Figure 4-27. BCU Waveforms

represents the MEM having been granted access to the busses. The FSM will stay in state two or three until the memory operation is complete. The VHDL code that implements the FSM is shown in Figure 4-29.



STATE	DESCRIPTION
0	BCU polling IF stage, IF granted busses if it requests them
1	BCU polling MEM stage, MEM granted busses if it requests them
2	IF stage has control of busses
3	MEM stage has control of busses

Figure 4-28. BCU FSM State Diagram

```
clk <= NOT clk AFTER 4 ns;

q0 <= ((NOT q1) AND (NOT q0) AND (NOT ir)) OR
      ((NOT q1) AND      q0  AND      mr ) OR
      (     q1  AND      q0  AND (NOT d )) OR
      (     q1  AND (NOT q0) AND      d  )
      WHEN clk'EVENT AND clk = '1' ELSE q0;

q1 <= ((NOT q1) AND (NOT q0) AND      ir ) OR
      ((NOT q1) AND      q0  AND      mr ) OR
      (     q1  AND (NOT d)                )
      WHEN clk'EVENT AND clk = '1' ELSE q1;
```

Figure 4-29. VHDL Code that Implements the FSM

5.0 STRUCTURAL MODEL

The structural model builds on the previous dataflow model. It uses the same dataflow test bench and top level components (memory module, compare module, and CPU component). The only modifications are to the ALU stage of the CPU component. The two units that are modeled at the structural level are the general register bank and the arithmetic logic unit block (ALUB), located in the ALU stage.

5.1 GENERAL PURPOSE REGISTER BANK

The general purpose register (GPR) bank is composed of thirty-two 32-bit general registers used by the processor for temporary storage for operands and results. The dirty bits for each register are also found in the GPR bank. The GPR bank component is shown in Figure 5-1. The *start* signal is generated from the *alu_start* signal in the ALU stage. It is only used to signal when to set the dirty bits. The *db_sel* signal is a 5-bit line that selects which register is to be set dirty. This signal comes from the ID stage. The *write* signal comes from the WB stage and signals when to write back the result to a register. It also signals when to clear the dirty bits. The *reg_sel* signal is a 5-bit line that selects

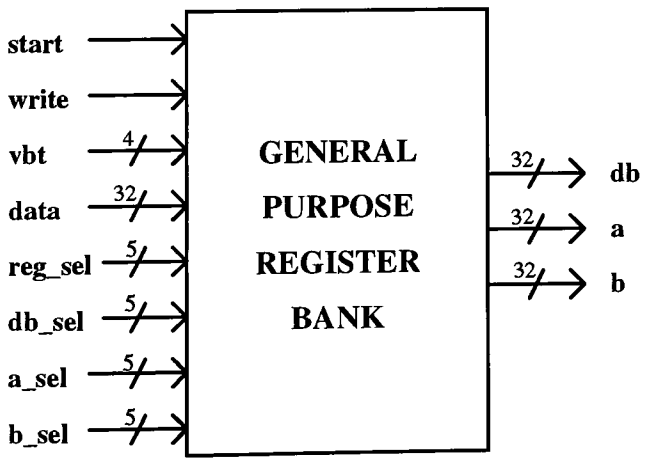


Figure 5-1. General Purpose Register Bank Component

which register is the destination register (which register is written back to). Once the register is updated with the result, it's dirty bit is cleared. The *data* signal is a 32-bit line that contains the data that is placed in the destination register. The valid byte tag (*vbt*) signal is a 4-bit line and is generated by the MEM stage. The *vbt* signal is used to determine what portion of the data is valid. The registers are logically divided into four bytes. Each bit of the *vbt* signal represents one of these bytes. Each byte of data is valid and can be written back when the corresponding *vbt* bit is high. The *db* signal is 32-bits wide and represents the 32 dirty bits, one for each register. The signal is used to stall the ID stage when a data dependency exists. The *a* and *b* signals are 32-bits wide and represent the *a* and *b* busses in the ALU stage. The *a_sel* and *b_sel* signals are 5-bits wide and select which register to put on the *a* and *b* busses, respectively. The components used in the GPR bank are shown in Figures 5-2, 5-3, and 5-4. Figure 5-2 is a transmission gate, Figure 5-3 is a 8-bit register, and Figure 5-4 is a 32-bit register. The GPR bank was designed by Kevin Johnson and a more detailed discussion can be found in "Design and Implementation of an Asynchronous Version of the MIPS R3000 Microprocessor" [10].

```
ARCHITECTURE sttg8_a OF sttg8 IS
BEGIN
    o <= i AFTER 0.1 ns WHEN en = '1' ELSE
         "ZZZZZZZZ" AFTER 0.1 ns;
END sttg8_a;
```

Figure 5-2. VHDL Code of an 8-bit Wide Transmission Gate

```
ARCHITECTURE streg8_a OF streg8 IS

    COMPONENT sttg8
        PORT(i:  IN  slv_8;
             en: IN  std_logic;
             o:  OUT slv_8);
    END COMPONENT;

    SIGNAL ld_bar: std_logic := '0';
    SIGNAL m1: slv_8 := "00000000";
```

```
    SIGNAL m2: slv_8 := "00000000";
    SIGNAL m3: slv_8 := "00000000";

BEGIN

    ld_bar <= NOT ld;

    input_tg:sttg8
        PORT MAP(i, ld, m1);

    m2 <= NOT m1 AFTER 0.3 ns;

    m3 <= NOT m2 AFTER 0.3 ns;

    feedback_tg:sttg8
        PORT MAP(m3, ld_bar, m1);

    a_bus_tg:sttg8
        PORT MAP(m3, as, a);

    b_bus_tg:sttg8
        PORT MAP(m3, bs, b);

    p <= m3;

END streg8_a;
```

Figure 5-3. VHDL Code of an 8-bit Register

```
ARCHITECTURE streg32_a OF streg32 IS

    COMPONENT streg8
        PORT(i:  IN  slv_8;
             ld: IN  std_logic;
             as: IN  std_logic;
             bs: IN  std_logic;
             a:  OUT slv_8;
             b:  OUT slv_8;
             p:  OUT slv_8);
    END COMPONENT;

    SIGNAL d0, d1, d2, d3: slv_8;
    SIGNAL ld0, ld1, ld2, ld3: std_logic;
    SIGNAL a0, a1, a2, a3: slv_8;
    SIGNAL b0, b1, b2, b3: slv_8;
    SIGNAL dbid, dbwb: std_logic := '0';
    SIGNAL p0, p1, p2, p3: slv_8;

BEGIN

    d0 <= data(7 DOWNTO 0);
    d1 <= data(15 DOWNTO 8);
    d2 <= data(23 DOWNTO 16);
    d3 <= data(31 DOWNTO 24);

    ld0 <= vbt(0) AND ld AFTER 1 ns;
    ld1 <= vbt(1) AND ld AFTER 1 ns;
```

```
    ld2 <= vbt(2) AND ld AFTER 1 ns;
    ld3 <= vbt(3) AND ld AFTER 1 ns;

    reg_byte0: streg8
        PORT MAP(d0, ld0, as, bs, a0, b0, p0);

    reg_byte1: streg8
        PORT MAP(d1, ld1, as, bs, a1, b1, p1);

    reg_byte2: streg8
        PORT MAP(d2, ld2, as, bs, a2, b2, p2);

    reg_byte3: streg8
        PORT MAP(d3, ld3, as, bs, a3, b3, p3);

    a <= a3 & a2 & a1 & a0;
    b <= b3 & b2 & b1 & b0;

    p <= p3 & p2 & p1 & p0;

    dbid <= dbid XOR '1' AFTER 1 ns
        WHEN db_set'EVENT AND db_set = '1'
        ELSE dbid;

    dbwb <= dbwb XOR '1' AFTER 1 ns
        WHEN ld'EVENT AND ld = '1'
        ELSE dbwb;

    db <= dbid XOR dbwb AFTER 1 ns;

END streg32_a;
```

Figure 5-4. VHDL Code of a 32-bit Register

5.2 ARITHMETIC LOGIC UNIT BLOCK

The ALUB consists of ten components: bus control block (BCB), arithmetic logic unit component (ALUC), ALUC decoder, overflow block, compare block, branch control, shifter unit, shifter unit control, set-on-less-than (SLT) unit, and output selector. The ALUB block diagram is shown in Figure 5-5.

BUS CONTROL BLOCK

The BCB, shown in Figure 5-6, controls what is placed on the *a* and *b* busses in the ALUB. These busses are fed into the ALUC. The BCB consists of the A-bus selector, the B-bus selector, and the bus selection decoder.

The A-bus selector, shown in Figure 5-7, consists of a 32-bit 4-to-1 multiplexer. This multiplexer selects from the following three input values: *a, pcl,* and the constant of zero. The *a* signal comes from the *a* output of the GPR bank. This signal is used when the instruction needs a value for the first source field (bits 25-21 of the instruction). The *a* output of the GPR bank supplies this first source field. The *pcl* signal is the current value of the PC and comes from the ID stage. This signal is used for jump-and-link instructions. It is used to calculate the link address that is stored in the link register (the *bcond* link instructions use the ADD8 unit to calculate the link address). The constant value of zero is needed for *shift* instructions. *Shift* instructions utilize the shifter unit and do not use the ALUC. Therefore, the data that is operated on has to be passed to the shifter unit unchanged. This data is stored in the general registers and is supplied on the *b* output of the GPR bank. The *a* input to the ALUC has to be zero so that when it is added to the *b* input it does not change. The *jump (j)* and *shift (s)* signals select which multiplexer input to use.

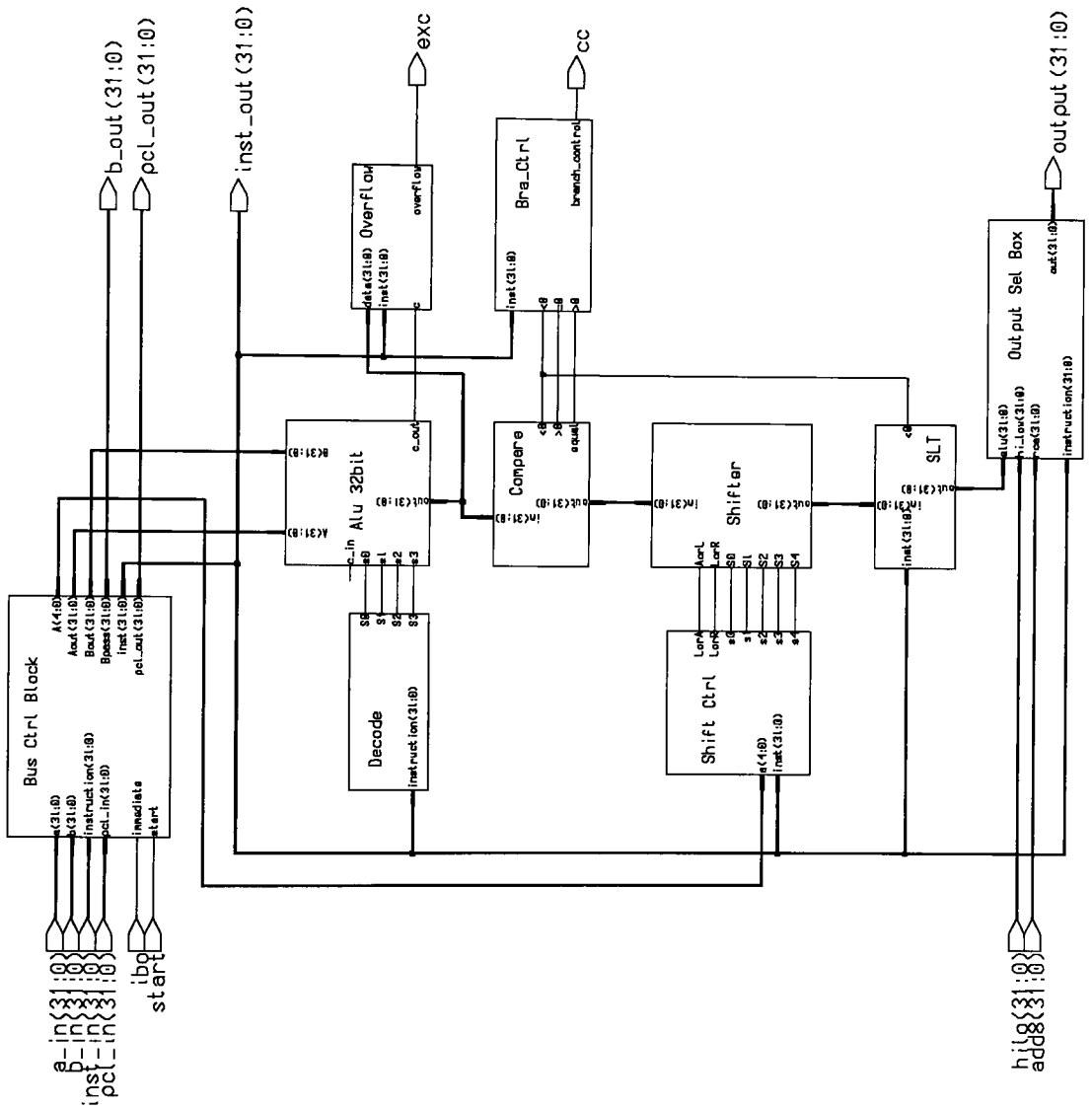


Figure 5-5. Arithmetic Logic Unit Block (ALUB) Diagram

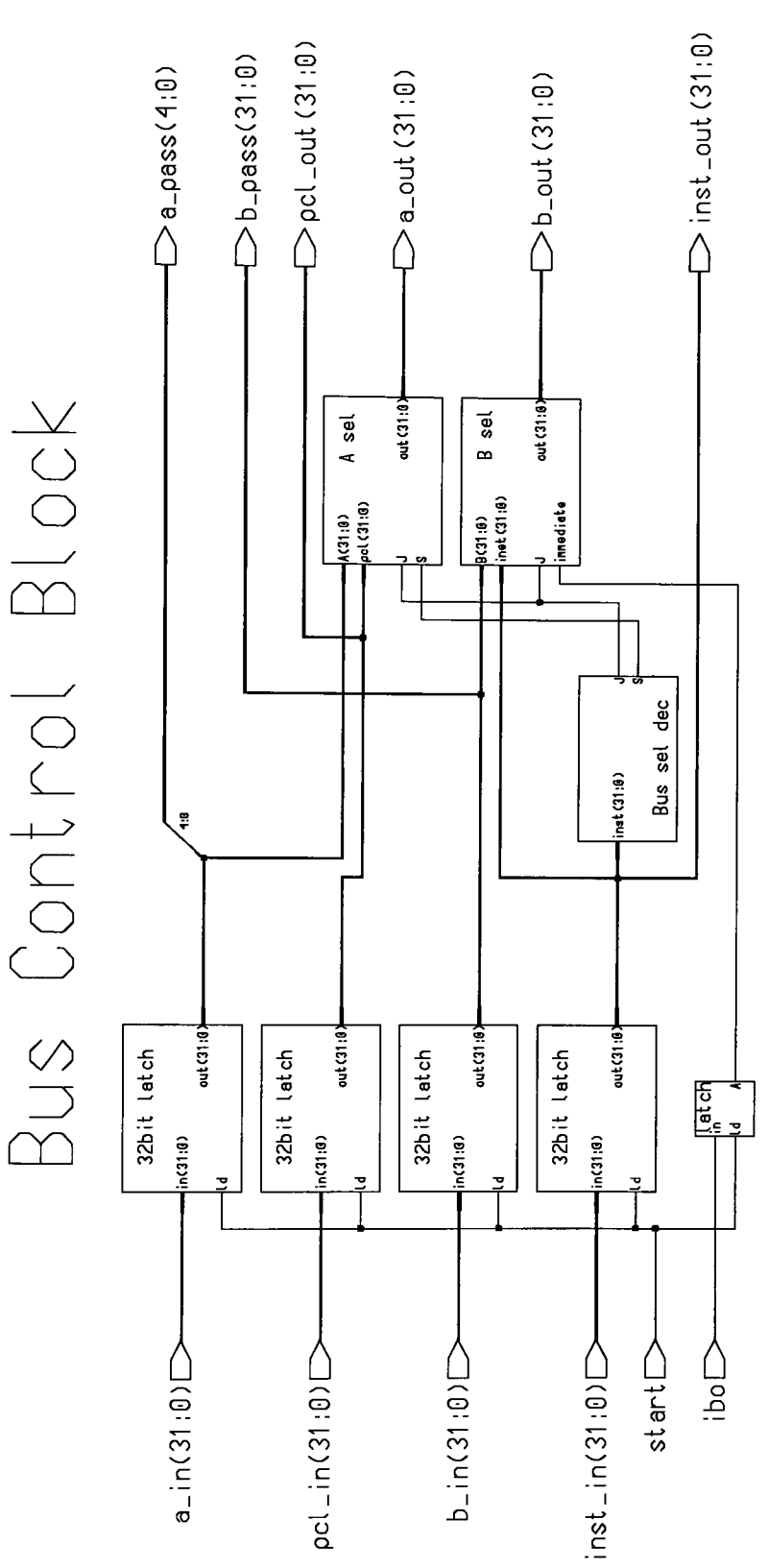


Figure 5-6. Bus Control Block (BCB) Diagram

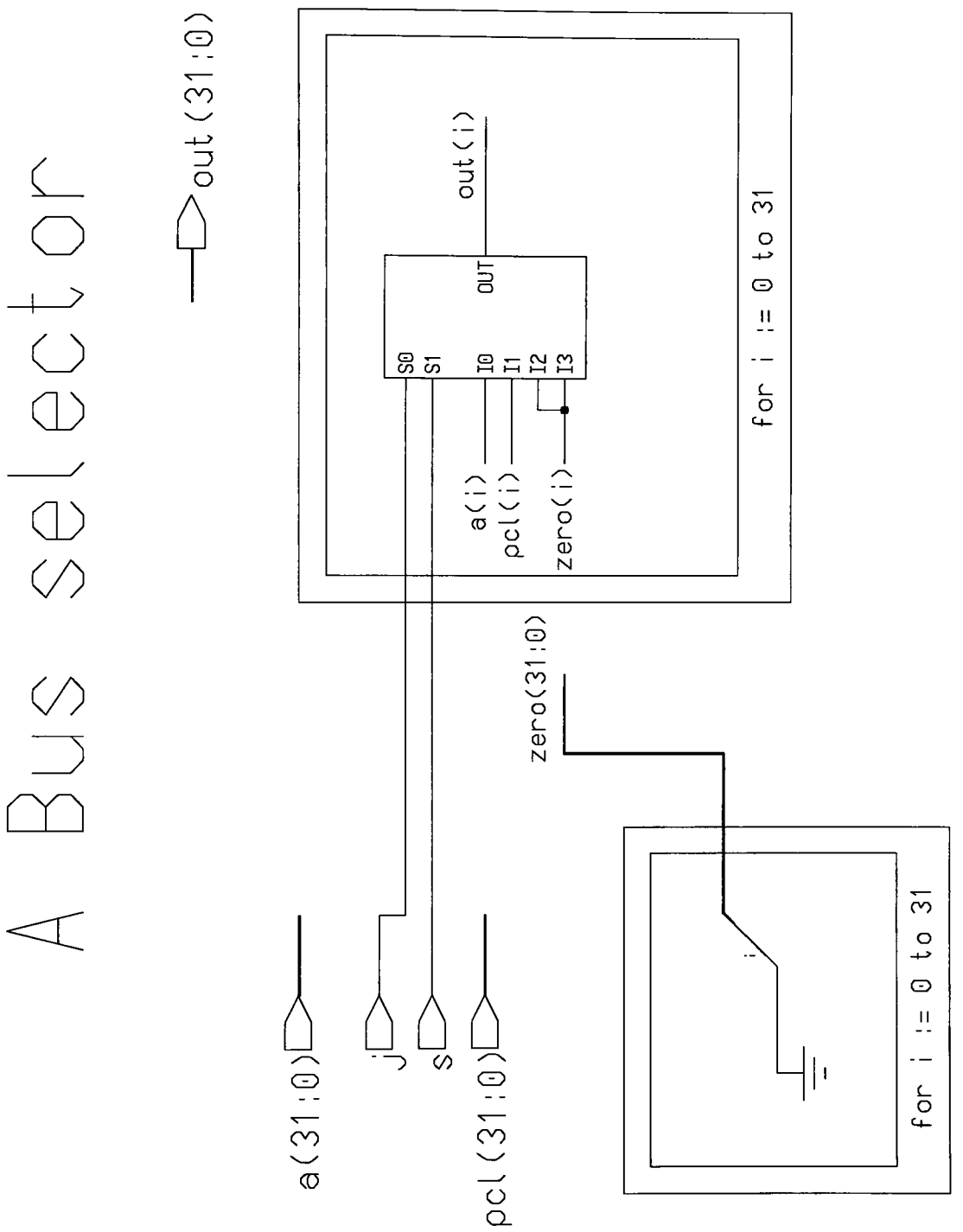


Figure 5-7. The A-Bus Selector Component

The B-bus selector, shown in Figure 5-8, also consists of a 32-bit 4-to-1 multiplexer and logic for immediate or base-offset instructions, sign-extension, and zero-extension. This multiplexer selects from the following three input values: *b, immed,* and the constant of eight. The *b* signal comes from the *b* output of the GPR bank. This signal is used when the instruction needs a value for the second source field (bits 20-16 of the instruction). The *b* output of the GPR bank supplies this second source field. The *immed* signal is the immediate field of the instruction (bits 15-0 of the instruction) that is sign or zero-extended to 32 bits. The immediate instructions that need to be sign-extended are *addi, addiu, slti,* and *sltiu.* The immediate instructions that need to be zero-extended are *andi, ori,* and *xori.* The *immed* input is chosen for an immediate instruction or for a base-offset calculation (here, the *immed* signal is the offset). The *load* and *store* instructions need the base-offset calculations. The constant value of eight is needed for link instructions. The link address is found by adding the value eight to the PC. The A-bus selector provides the PC to the *a* input of the ALUC. The B-bus selector provides the constant eight. The *jump (j)* and *ibo* signals select which multiplexer input to use.

The bus selection decoder, shown in Figure 5-9, provides the A-bus and B-bus selectors with the multiplexer select lines. The bus selection decoder generates the *j* select line when the instruction is either a *jump and link (jal)* or *jump and link register (jalr).* It also generates the *s* select line for all *shift* instructions.

ARITHMETIC LOGIC UNIT COMPONENT

The ALUC, is the component that does the addition, subtraction, and logical calculations. The ALUC also does base-offset address calculations. The component, shown in Figure 5-10, consists of eight ports. The *a* and *b* ports are 32 bits wide and correspond to the *a* and *b* outputs of the BCB. The four select lines, *s0* through *s3*,

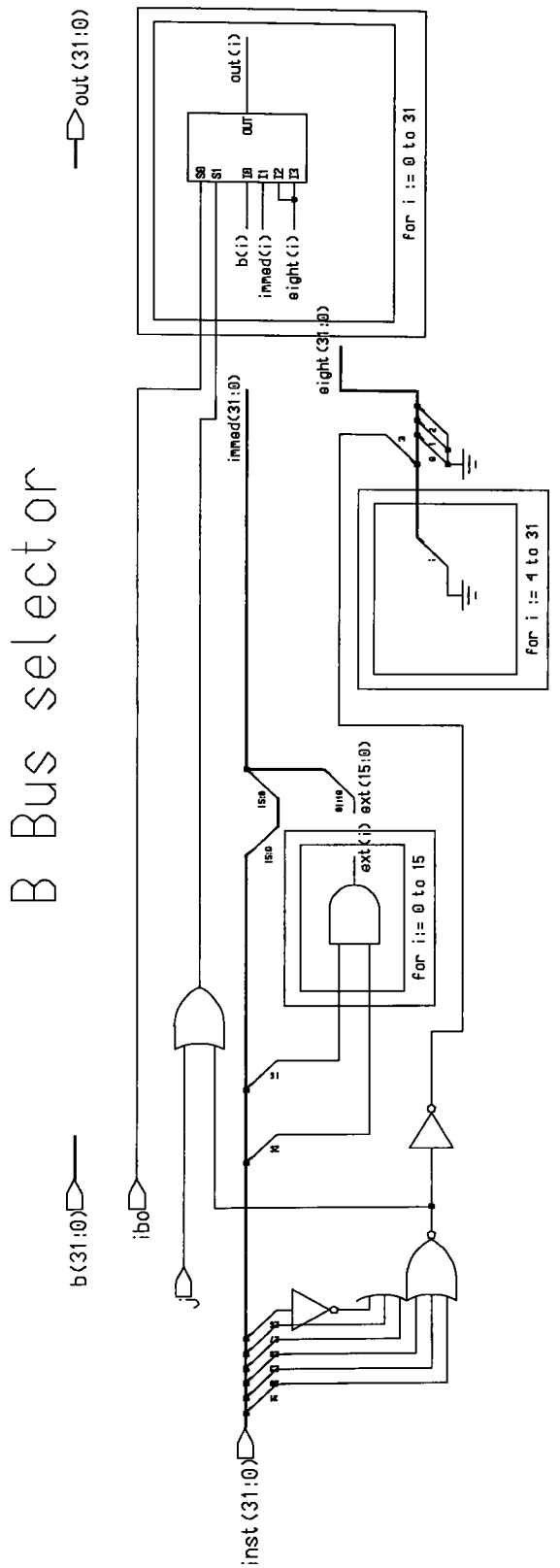


Figure 5-8. The B-Bus Selector Component

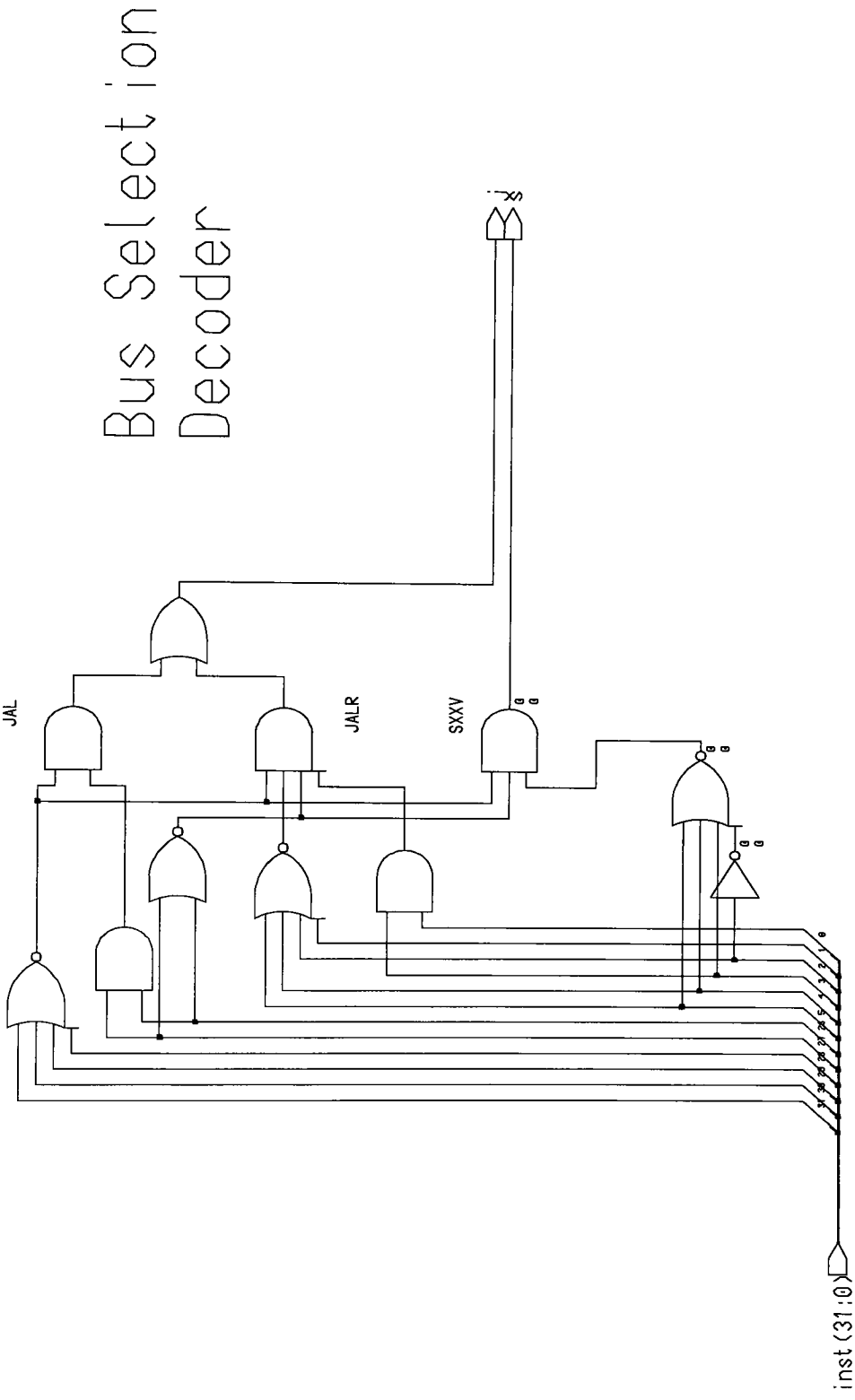


Figure 5-9. The Bus Selection Decoder Component

control what operation the ALUC performs. Table 5-1 lists these operations. The *out* signal is the output of the ALUC. The last port, *c_out*, is the carry output of the ALUC. The ALUC for the asynchronous version could not be modeled exactly after the R3000's ALUC. This was due to a lack of information on the R3000's internal gate design. However, a suitable design was found in the Fairchild Advanced Schottky TTL Data Book. More details can be found in Kevin Johnson's thesis [10].

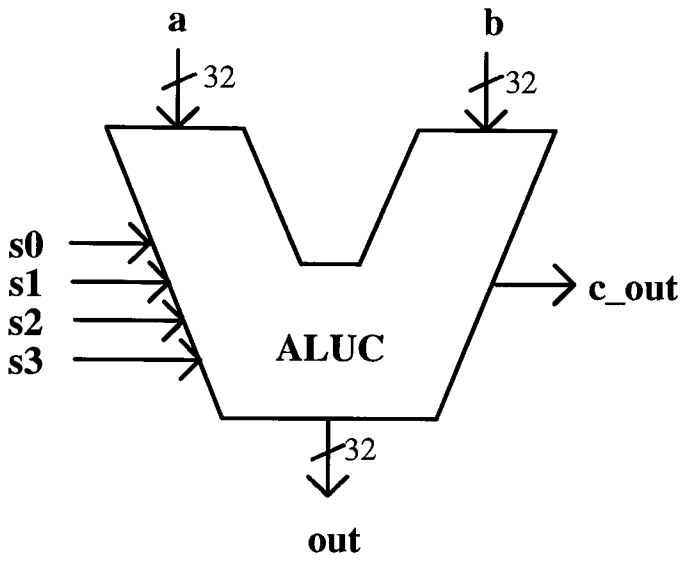


Figure 5-10. Arithmetic Logic Unit Component (ALUC)

OPERATION	S0	S1	S2	S3
a MINUS b	0	1	0	0
a PLUS b	1	1	0	0
a XOR b	0	0	1	0
a OR b	1	0	1	0
a AND b	0	1	1	0
a NOR b	0	1	1	1

Table 5-1. ALUC Operation Encoding

ALUC DECODER

Since the details of the synchronous R3000's ALUC were not known, the asynchronous version ALUC control signals do not match the instruction encoding inherent in the R3000. The ALUC decoder is a circuit that is required to decode the instruction before the ALUC can operate on it. This circuit is shown in Figure 5-11. A more detailed discussion on the ALUC decoder can be found in [10].

OVERFLOW BLOCK

The overflow block component, shown in Figure 5-12, detects an overflow condition from the ALUC. The three instructions that check for an overflow condition are *addi, add,* and *sub*. The carry signal, *c*, from the ALUC is XORed with the bit 31 of the *data* signal. The *data* signal is the output of the ALUC.

COMPARE BLOCK

The compare block component, shown in Figure 5-13, compares the output of the ALUC with the value of zero. It is used in conjunction with the branch control unit, discussed in the next section. The three output ports, *ltz, gtz,* and *eqz*, correspond to the three conditions: less than zero, greater than zero, and equal to zero.

BRANCH CONTROL

The branch control component, shown in Figure 5-14, determines if a branch is taken. The three inputs *ltz, eqz,* and *gtz*, are generated by the compare component discussed above. The *inst* input is used to determine the branch instruction type. The four

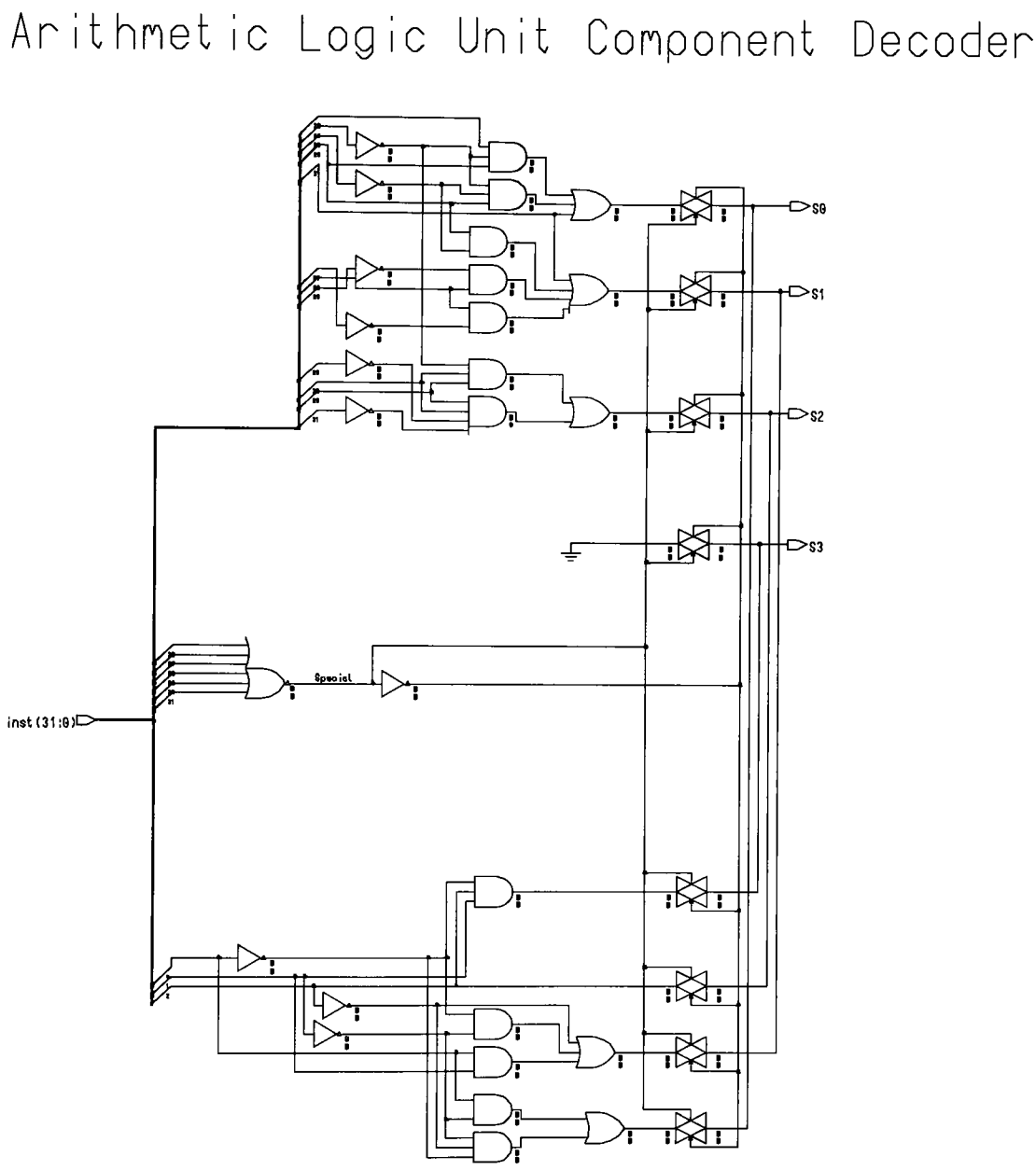


Figure 5-11. The Arithmetic Logic Unit Component (ALUC) Decoder

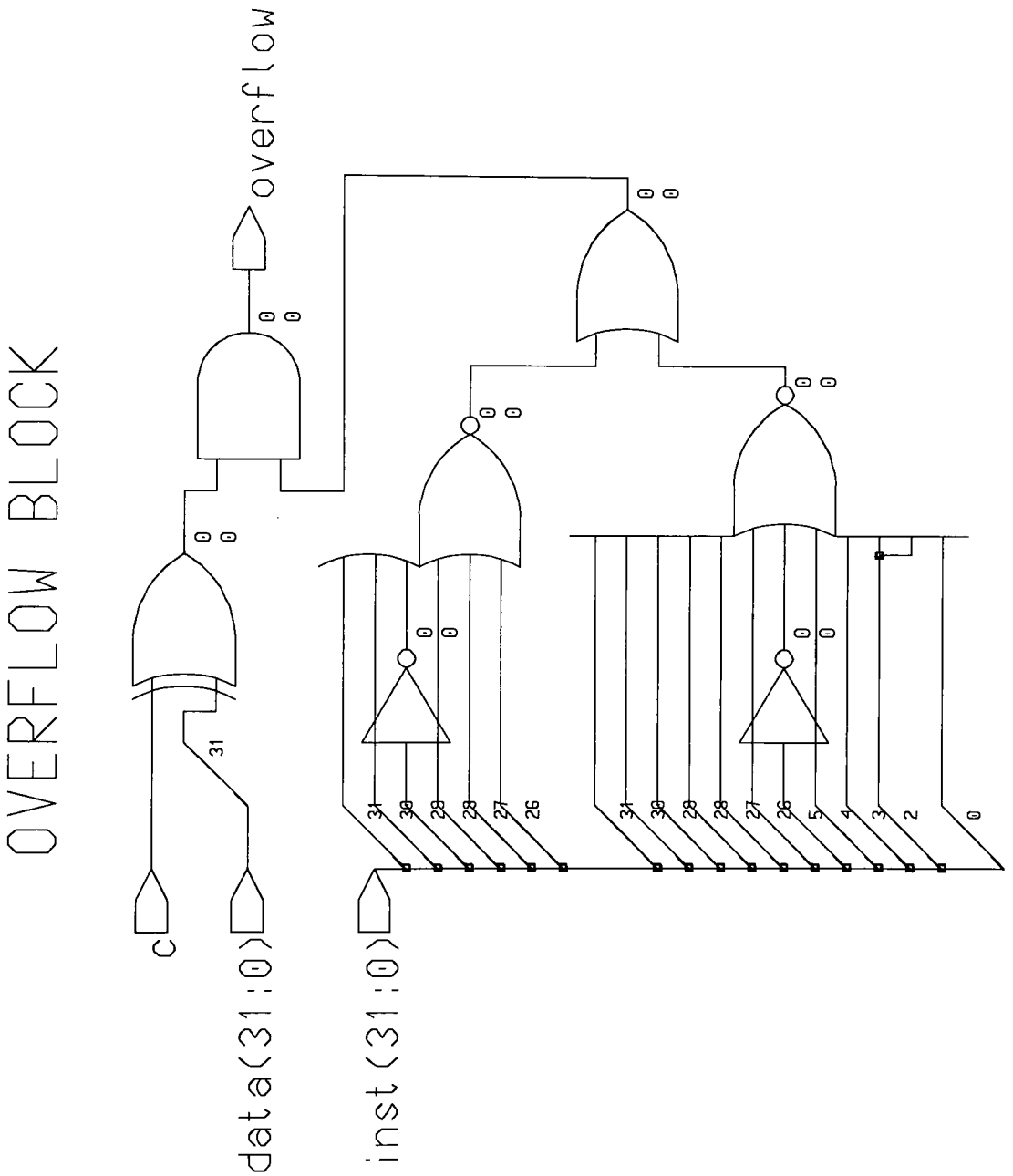


Figure 5-12. The Overflow Block Component

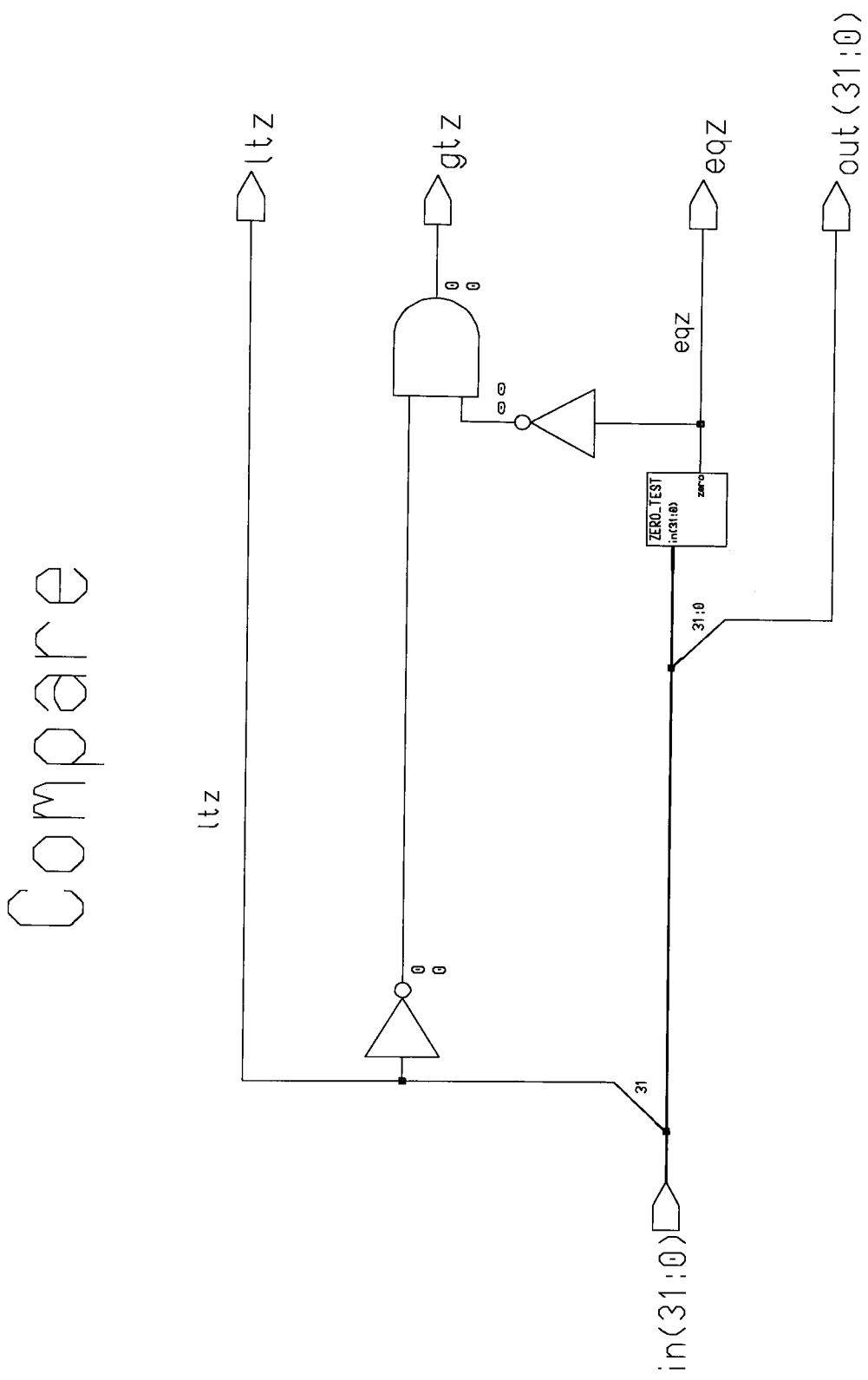


Figure 5-13. The Compare Block Component

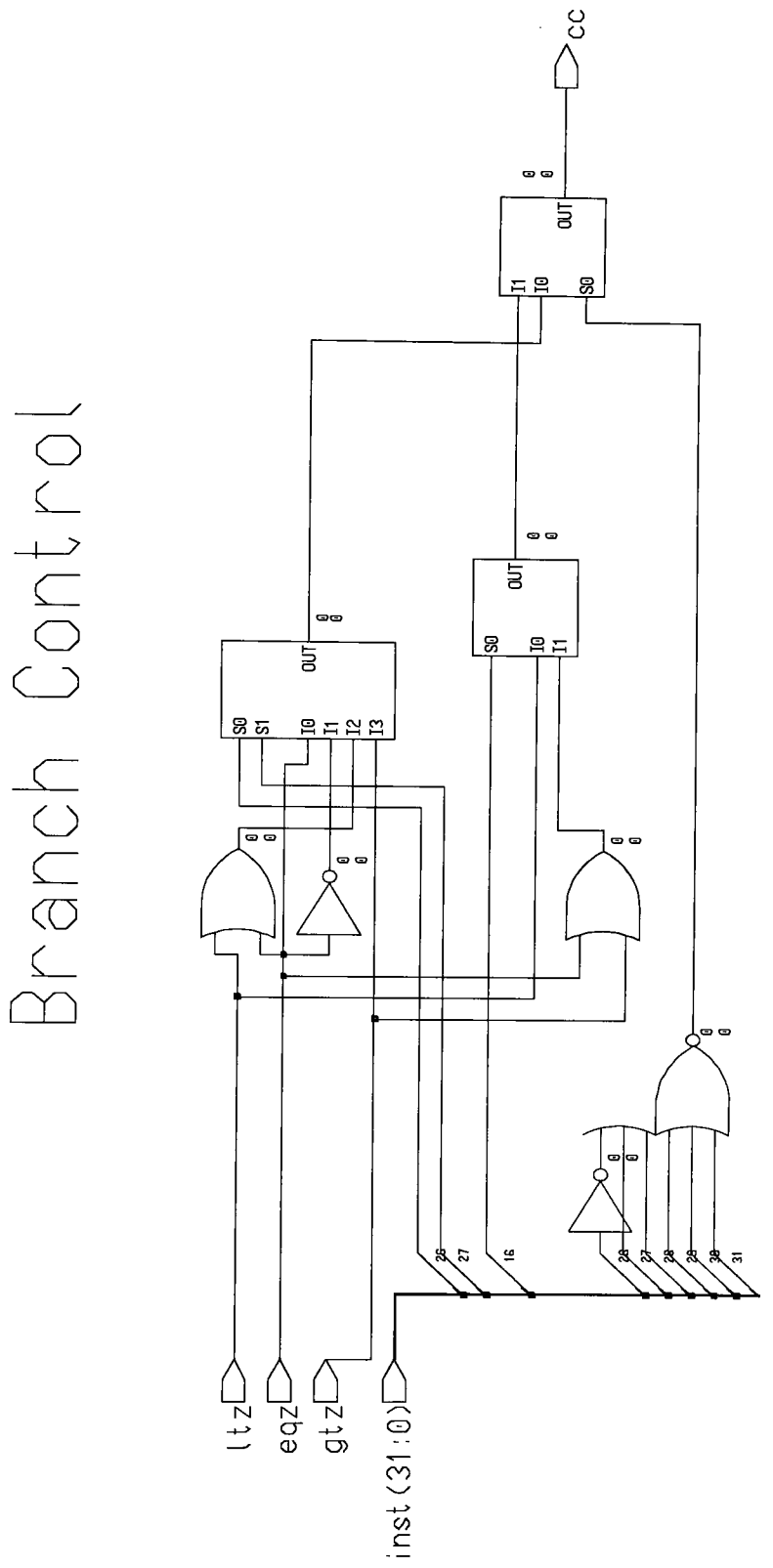


Figure 5-14. The Branch Control Component

regular branch instructions are branch on equal (beq), branch on not equal (bne), branch on less than or equal to zero (blez), and branch on greater than zero (bgtz). The four branch conditional (bcond) instructions are branch on less than zero (bltz), branch on greater than or equal to zero (bgez), branch on less than zero and link (bltzal), and branch on greater than or equal to zero and link (bgezal). The branch control output, cc, is the condition code and is high when the branch is taken.

SHIFTER UNIT

The shifter unit component, shown in Figure 5-15, provides the microprocessor with arithmetic and logical shift operations. The *lr* signal controls the direction of the shift (left or right). The *la* signal controls the type of shift (logical or arithmetic). The five selector bits, *s0* through *s4*, control the amount of bits to shift. The *in* and *out* signals correspond to the input and output of the shifter unit, respectively. A more complete discussion of the shifter unit can be found in Kevin Johnson's Thesis [10].

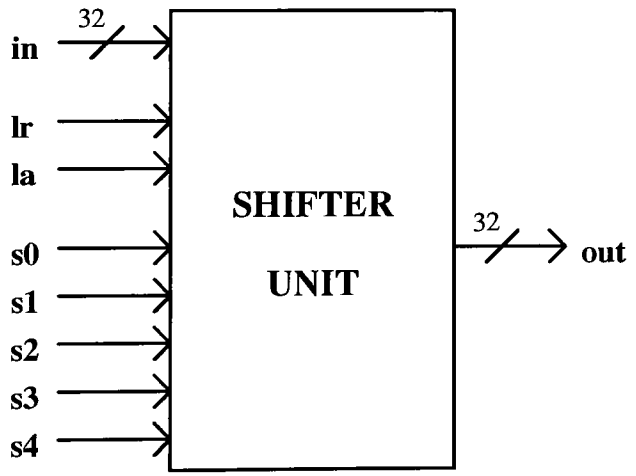


Figure 5-15. The Shifter Unit Component

SHIFTER UNIT CONTROL

The shifter unit control component, shown in Figure 5-16, provides the necessary signal inputs to the shifter. It generates the *lr, la,* and the five shift amount select lines (*s0* through *s4*). The *a* input is used in conjunction with the variable shifting instructions, *shift left logical variable (sllv), shift right logical variable (srlv),* and *shift right arithmetic variable (srav).* The variable instructions have their variable shift field in bits 25-21 of the instruction. This shift field is placed on the A-bus. The *a* input is the lower five bits of the A-bus from the BCB.

SET ON LESS THAN UNIT

The set on less than unit component, shown in Figure 5-17, implements the four instructions, *set on less than (slt), set on less than unsigned (sltu), set on less than immediate (slti),* and *set on less than immediate unsigned (sltiu).* It consists of a 1-bit multiplexer, a 31-bit tri-state buffer, and decoding logic. All *slt* instructions output a value of one when the first source register (*rs*) is less than the second source register (*rt*). Otherwise they output a zero. For a *slt* instruction, the tri-state buffers are activated and the top 31 bits are set to zero. Bit zero is determined by the less than zero (*ltz*) signal. For non *slt* instructions, the tri-state buffers are deactivated and the input (*in*) passes to the output (*out*) unchanged.

OUTPUT SELECTOR

The output selector component, shown in Figure 5-18, is the final component in the ALUB. Its task is to route the proper lines to the ALUB *output* port. The output selector chooses from the following three inputs: *alu, hilo,* or *add8.* The *alu* signal input

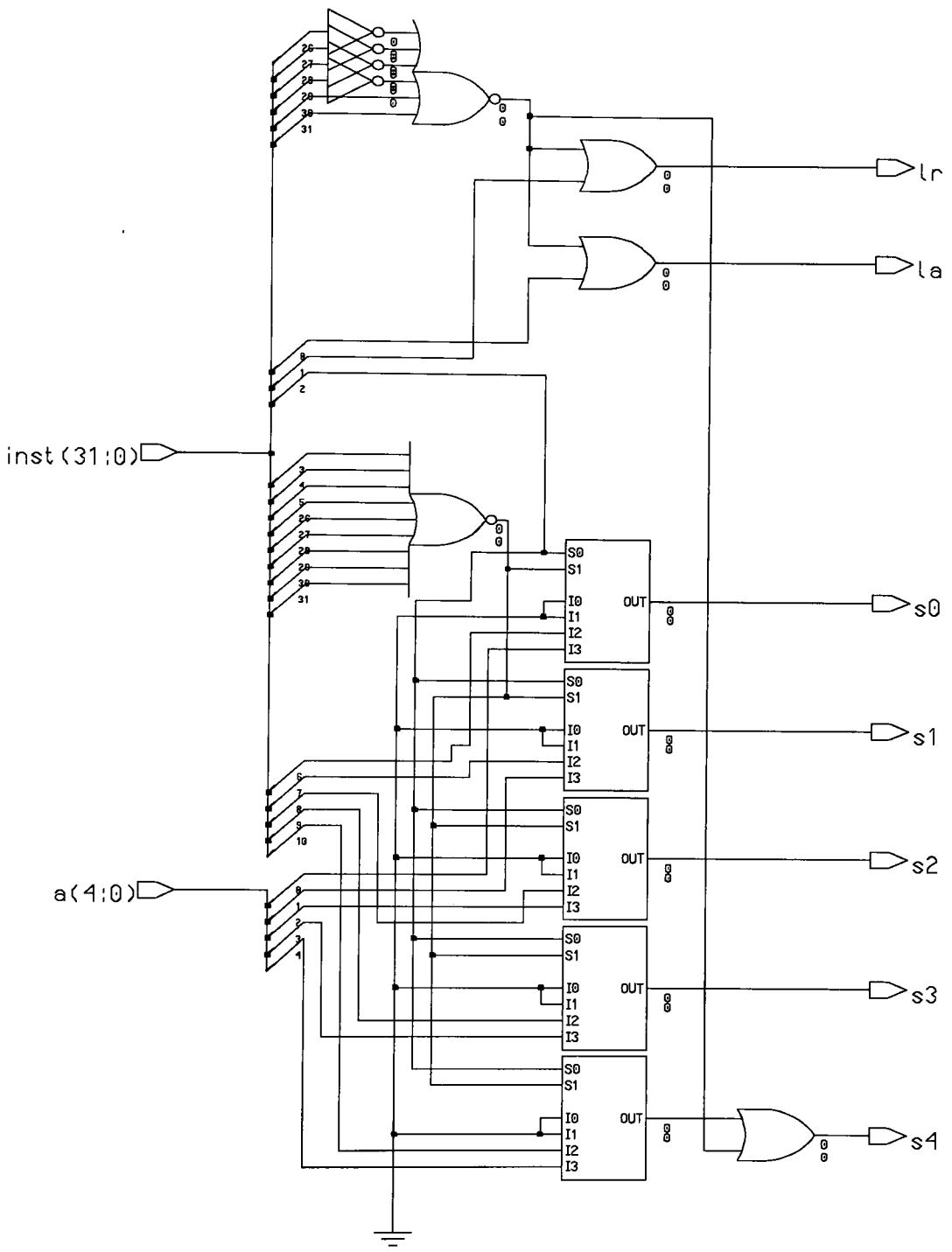


Figure 5-16. The Shifter Unit Control Component

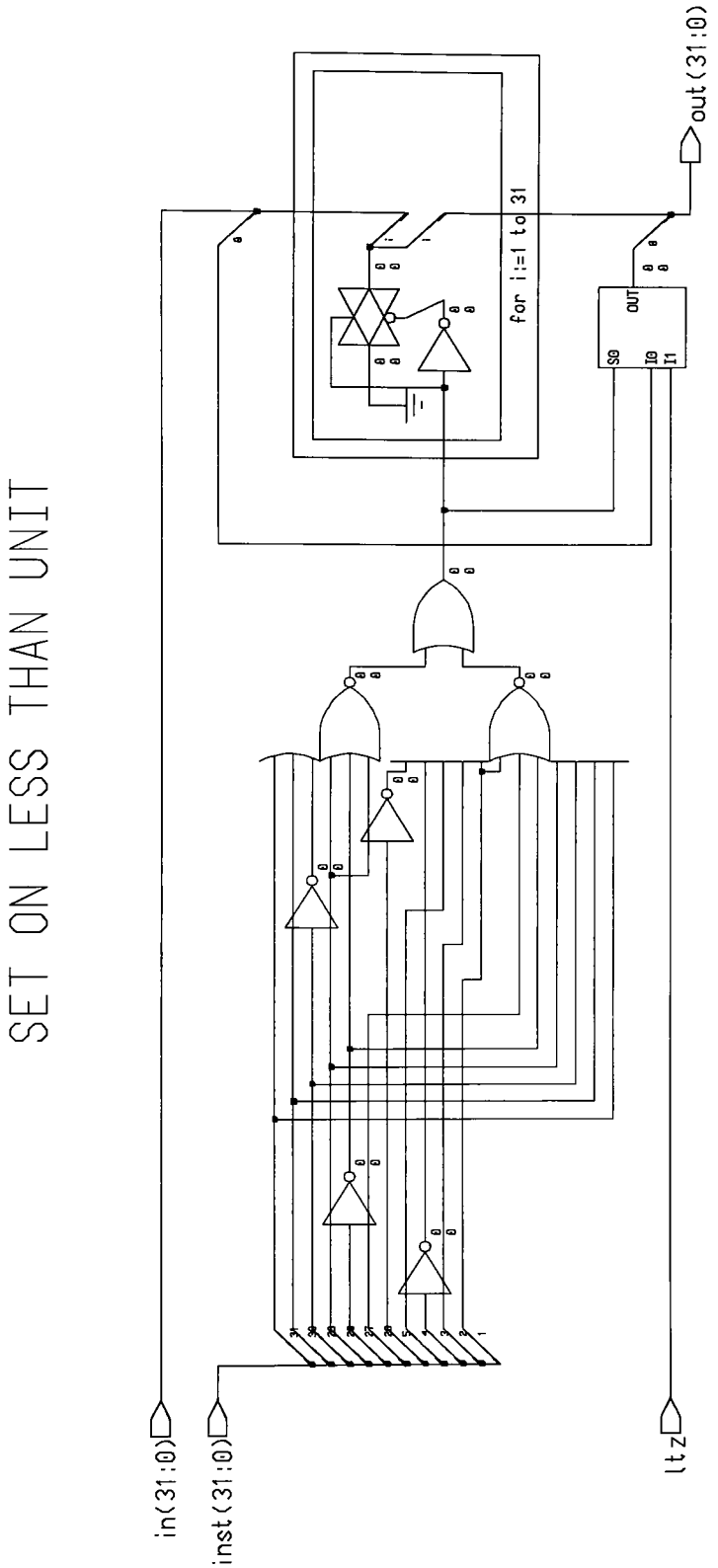


Figure 5-17. The Set On Less Than Unit Component

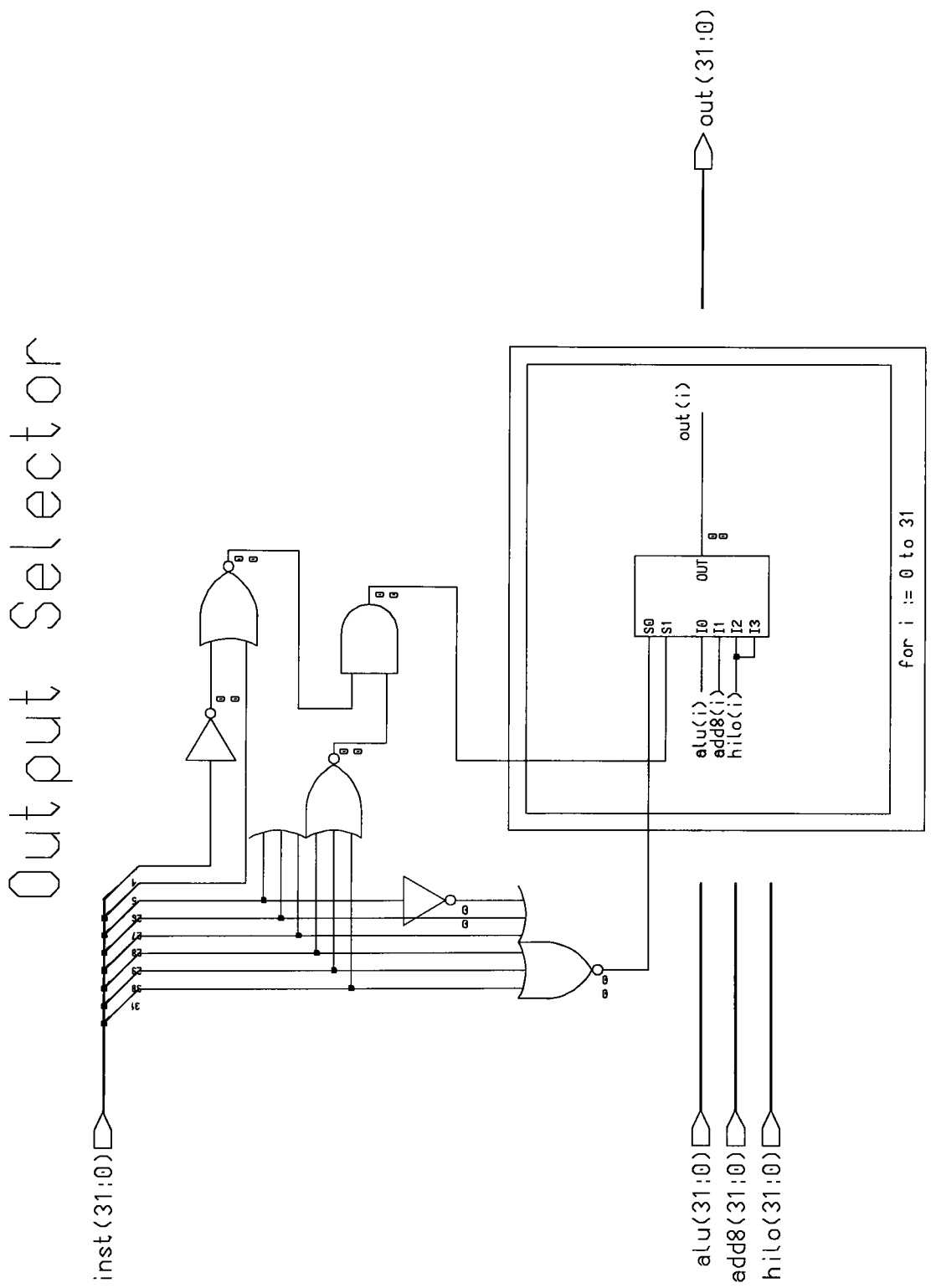


Figure 5-18. The Output Selector Component

comes from the output of the component chain in the ALUB. The *hilo* signal is used for the *move from hi (mfhi)* and *move from lo (mflo) instructions*. The *add8* signal is used for the two *branch conditional link* instructions, *branch on less than zero and link (bltzal)* and *branch on greater than or equal to zero and link (bgezal)*.

6.0 RESULTS

All three models were tested using a VHDL test bench. The test bench setup is shown in Figure 1-1. The behavioral model used the test bench shown in Figure 3-1. The dataflow and structural models used the test bench shown in Figure 4-1. Both test benches consists of three modules, CPU, memory, and compare. The memory module stores the test program used in the test bench. The compare module stores the expected results file. The output of the CPU module is compared against the expected results file after each instruction is executed.

6.1 TESTING PROCEDURE

The testing process consists of writing the test program, assembling the test program, generating an expected results file, processing the expected results file, preprocessing the test program and expected results file, loading the model into the digital simulator, and running the model. The model runs to completion if there are no discrepancies between the model and expected results. However, the user is warned if a discrepancy exists.

The test program can be written using any text editor. The programs are written in a pseudo MIPS assembly language. The program is saved with a "test" extension. An example of a filename is "*filename*.test" where *filename* is the name of the file.

The test program is assembled by the MIPS assembler (MASS) program, found in Appendix D. This assembler generates the machine code that the model can understand. The generated file is saved with a "m" extension which stands for the word machine. An example of this is "*filename*.m" where *filename* is the same name used as above.

The expected results file is generated using the MIPS expected results assembler (MERA) program, found in Appendix D. The user inputs the expected state of the

processor after every instruction into the assembler. This file is then used by the compare module. The file is saved with an "e" extension which stands for the word expected. An example of this is "*filename*.e".

The expected results file has to be modified for test programs with branches and jumps. The FLOW program was written to reorder such programs to follow the actual program flow. The instructions in the test program can now be compared to the reordered expected results file. The reordered expected results file is saved with a "f" extension. An example of this is "*filename*.f". The original expected results file, "*filename*.e", is unchanged.

The test files are preprocessed with the MIPS preprocessor (MPP) program. The preprocessor prepares the test files so the model can use them. The MPP program copies the appropriate test files to two files that the model opens. These two files are called "machine" and "expected" which represent the machine code file loaded into the memory module and the expected results file loaded into the compare module, respectively. The MPP program copies "filename.m" to "machine" and "filename.e" to "expected".

The model is loaded into the digital simulator. The name of the digital simulator is Quicksim and is manufactured by Mentor Graphics Corporation. The two test files, "machine" and "expected", need to be in the same directory as the model's design file. Quicksim is also invoked from this directory.

Running the model in Quicksim consists of setting up the simulator, setting some signal values, and running the simulator for some set time. Setting up the simulator involves opening the VHDL code windows, tracing pertinent signals, and opening list and monitor windows. When the signals are forced to specific values, they can be traced on the screen. The *sys_control_sig* signal is used to control the model. This signal is forced to a value of *load* for 100 nanoseconds (ns) and then forced to a value of *run*. The *load* value mode signals the test bench to load the test program ("machine" file) into the

memory module and the expected results file ("expected" file) into the compare module. The *run* value mode signals to the test bench to run the model for testing.

The models use VHDL ASSERT statements to alert the user if there is a problem with the simulation. If there are no problems and the state of the processor matches the expected results, then the model will run to completion. The user is only warned when an error or discrepancy exists.

The test benches were used throughout the entire design process. During the behavioral modeling as the instructions were coded, the test bench was used to test whether the model was performing up to specifications. During the dataflow modeling, the test bench was used first to verify the arbitrary simulation times and then used again for the back annotated times from the Accusim circuit simulation runs. All models were verified using their appropriate test benches.

6.2 DELAY TIMES

This section shows the gate, component and stage delay times that were back annotated into the model from Accusim circuit simulations. These times were taken from Kevin Johnson's thesis [10] and Scott Siers' thesis [19]. These circuit simulations are found in their separate documents.

GATE DELAY TIMES

Circuit simulations were performed on the individual gates. Table 6-1 shows these times. Delay times of complex circuits that are made up of these gates were calculated by determining the critical path and then adding up the times.

GATE	DELAY TIME (ns)
inverter	0.3
NAND	0.7
NOR	1.0
AND	1.0
OR	1.3

Table 6-1. Gate Delay Times

COMPONENT DELAY TIMES

Primitive components delay times were calculated by simulating the component. More complex components are made up of primitive components. Their delay times were calculated by determining the critical path and adding up the times. This provides a worst case value. Table 6-2, shown below, shows the various component delay times used.

COMPONENT FILENAME	COMPONENT	DELAY TIME (ns)
df2to1mux	1-bit 2 to 1 mux	0.3
df2to1mux16	16-bit 2 to 1 mux	0.3
df2to1mux3	3-bit 2 to 1 mux	0.3
df2to1mux30	30-bit 2 to 1 mux	0.3
df2to1mux32	32-bit 2 to 1 mux	0.3
df2to1mux8	8-bit 2 to 1 mux	0.3
df32to1mux	1-bit 32 to 1 mux	3.6
df4to1mux	1-bit 4 to 1 mux	0.4
df4to1mux10	10-bit 4 to 1 mux	0.4
df4to1mux16	16-bit 4 to 1 mux	0.4
df4to1mux32	32-bit 4 to 1 mux	0.4
df4to1mux4	4-bit 4 to 1 mux	0.4
df4to1mux8	8-bit 4 to 1 mux	0.4
dfaa	address adder	8
dfadd8	add8 unit	26
dfalu	ALU stage	see Table 6-3
dfalu32	ALU component (ALUC)	10
dfalublk	ALU block (ALUB)	30.1

dfaludec	ALU decoder	2.5
dfasel	A-bus selector	0.4
dfbc	bus controller	internal clock used for FSM high speed poller: 4
dfbctl	branch control	2
dfbjbox	branch and jump box	2.1
dfbsel	B-bus selector	2.0
dfbusctl	bus control block	6.2
dfbusdec	bus selection decoder	3.3
dfcomp	compare block	4.4
dfcompare	test bench compare module	N/A - (behavioral)
dfcpu	test bench CPU module	see Table 6-3
dfdbox	dirty box	3.1
dfeh	exception handler	1.5
dffed	falling edge detector	delay time: 0.4 pulse width: 1.2
dfhcc	handshaking control circuit	5.7
dfhlreg	hi/lo register bank	write: 0.3 read: 0.9
dfid	ID stage	see Table 6-3
dfif	IF stage	see Table 6-3
dfinstdec	ID instruction decoder	4.4
dfmd	multiplier/divider	1005
dfmem	MEM stage	see Table 6-3
dfmemdec	MEM stage decoder	5.3
dfmemory	test bench memory module	memory speed: 50 handshaking overhead: 5 total delay time: 55
dfmu	mask unit	3.4
dfoutsel	output selector	2.7
dfovrf	overflow block	4.6
dfred	rising edge detector	delay time: 0.3 pulse width: 1.4
dfreg	1-bit register	0.9
dfreg32	32-bit register	0.9
dfreg4	4-bit register	0.9
dfreg5	5-bit register	0.9
dfregbank	thirty-two 32-bit general purpose registers	write: 0.3 read: 0.9
dfregr	1-bit register with reset	0.9 0.3 for reset
dfrslat	set-reset latch	set time: 1.0 reset time: 0.4

dfsctl	shifter unit control	2.7
dfshift	shifter unit	3.3
dfslt	set on less than unit	3.5
dfsu	shift unit	1.4
dftrds	target register dirty select	0.4
dftsb32	32-bit tri-state buffer	0.3
dfwb	WB stage	see Table 6-3

Table 6-2. Component Delay Times

STAGE DELAY TIMES

The stage delay times vary depending on the instruction executed. Table 6-3 shows the delay times for each pipeline stage for various instructions. The times of all the stages are added together to obtain the time that it takes to execute a particular instruction type. This is the CPU processing time. The HCC overhead is taken into account.

	DELAY TIMES (ns)				
STAGE	**ALU REG**	**BRANCH**	**JUMP**	**LOAD**	**MULT**
IF	69	78	69	150	69
ID	15	15	15	17	15
ALU	35	35	4	35	35
MEM	11	11	11	155	11
WB	9	9	9	9	9
CPU *	162	171	131	413	162

* HCC overhead for each stage included

Table 6-3. Stage Delay Times

6.3 EXAMPLE SIMULATIONS

This section consists of five examples, each corresponding to the instruction type in Table 6-3. The examples are taken from a few instructions of the test programs found

in Appendix E. The times shown in Table 6-3 came from these example simulations. The five instruction groups are as follows: *ALU register, branch, jump, load,* and *multiplication.*

The first example is taken from instructions 4, 5, and 6 of the arithmetic register test program ("ar.test"). Instruction 4 (PC equal to '0C') loads register 1 with the value 'A'. Instruction 5 loads register 2 with the value '5'. Instruction 6 adds registers 1 and 2 together and then places the result in register 3. The waveforms are shown in Figure 6-1.

The second example is taken from instructions 38, 39, and 40 of the jump and branch test program ("jb.test"). Instruction 38 (PC equal to '94') loads register 2 with the value 'ABCD'. Instruction 39 loads register 3 with the value 'ABCD'. Instruction 40 compares instructions 38 and 39, and branches to the destination address (5 instructions after the delay slot) if they are equal. The waveforms are shown in Figure 6-2.

The third example is taken from the first instruction of the jump and branch test program ("jb.test"). Instruction 1 (PC equal to '00') jumps to the destination address unconditionally with a delay of one instruction. The destination address is calculated by shifting the 26-bit target address left two bits and combining it with the high order 4 bits of the PC. The waveforms are shown in Figure 6-3.

The fourth example is taken from instruction 21 of the load and store test program ("ls.test"). Instruction 21 (PC equal to '50') loads register three with the contents of memory location '4000'. This value will be available for use after a delay of one instruction. The waveforms are shown in Figure 6-4.

The fifth and final example is taken from instructions 1, 2, and 3 of the multiplication and division test program ("md.test"). Instruction 1 (PC equal to '00') loads register 1 with the value '8'. Instruction 2 loads register 2 with the value '9'. Instruction 3 multiplies register 1 and 2 together and places the results in the *hi* and *lo* registers. The waveforms are shown in Figure 6-5.

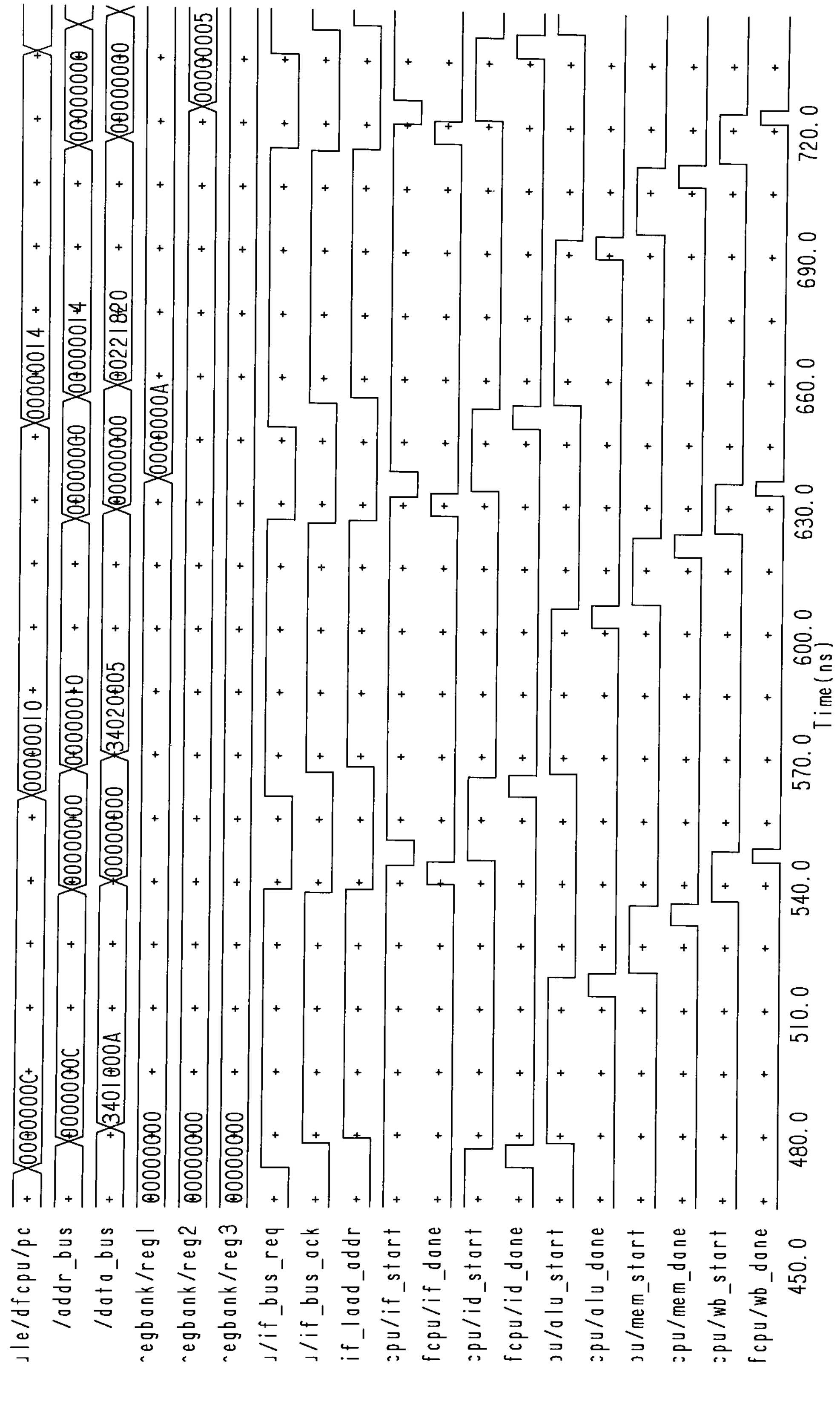


Figure 6-1. Example 1 - Arithmetic Register Instruction Waveforms

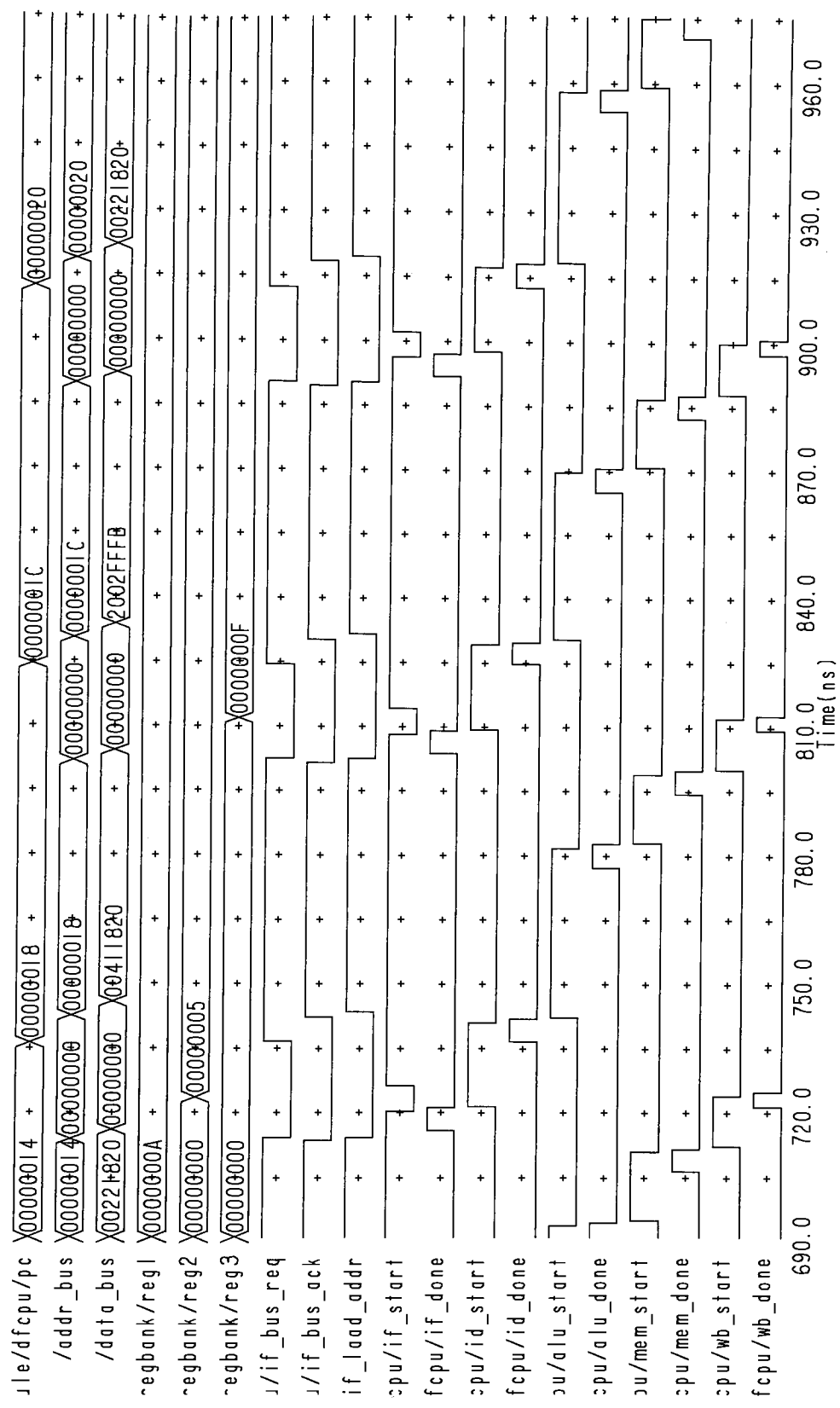


Figure 6-1 Continued

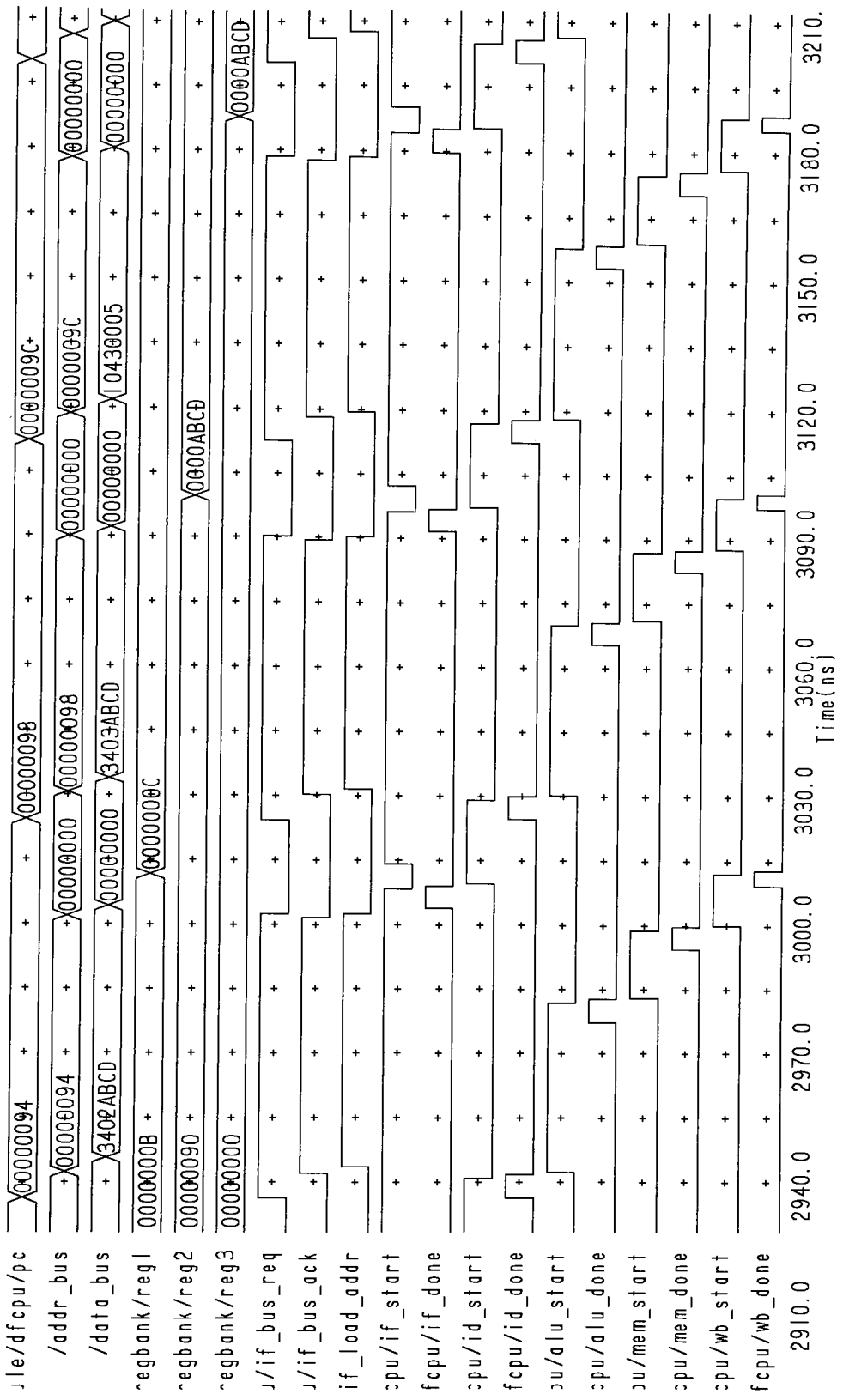


Figure 6-2. Example 2 - Branch Instruction Waveforms

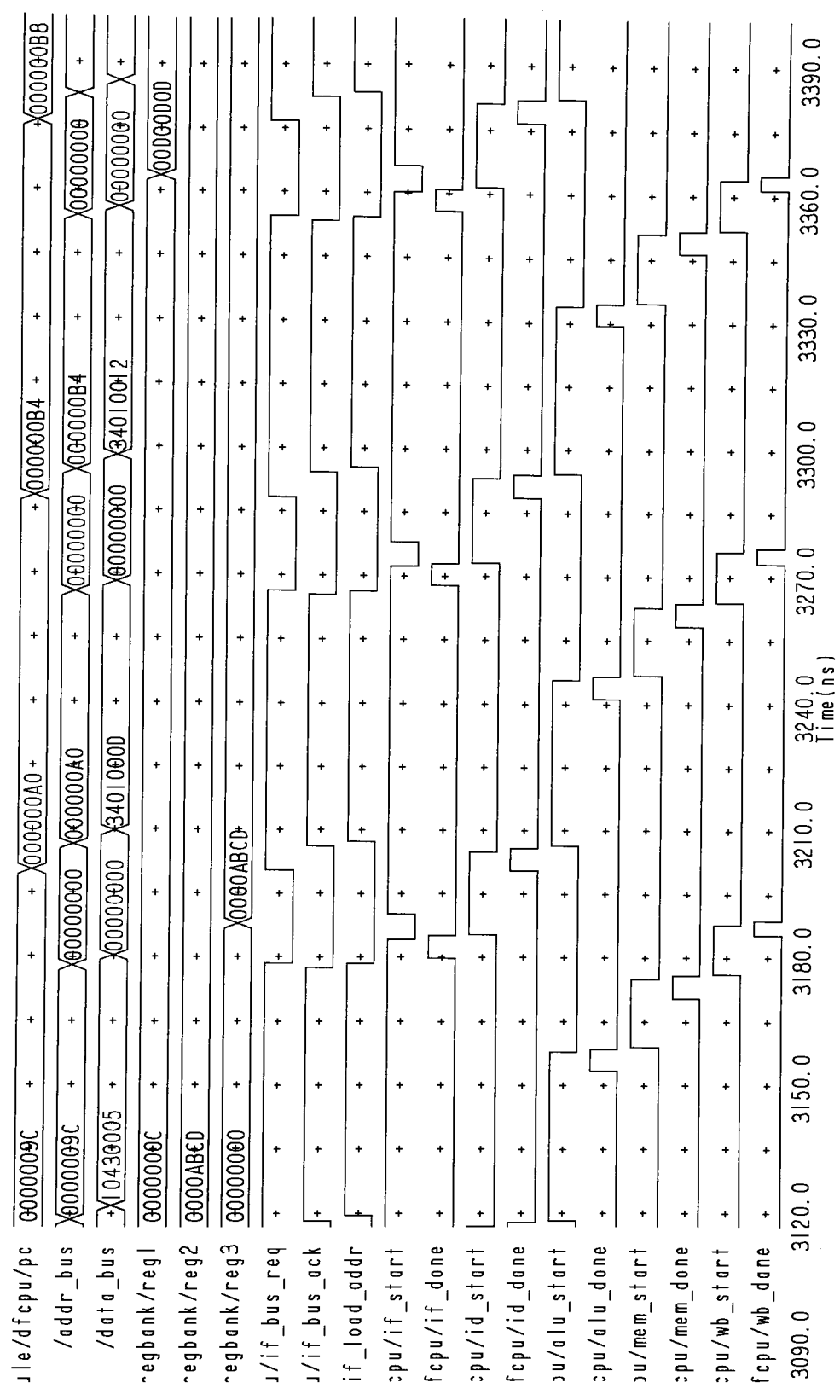


Figure 6-2 Continued

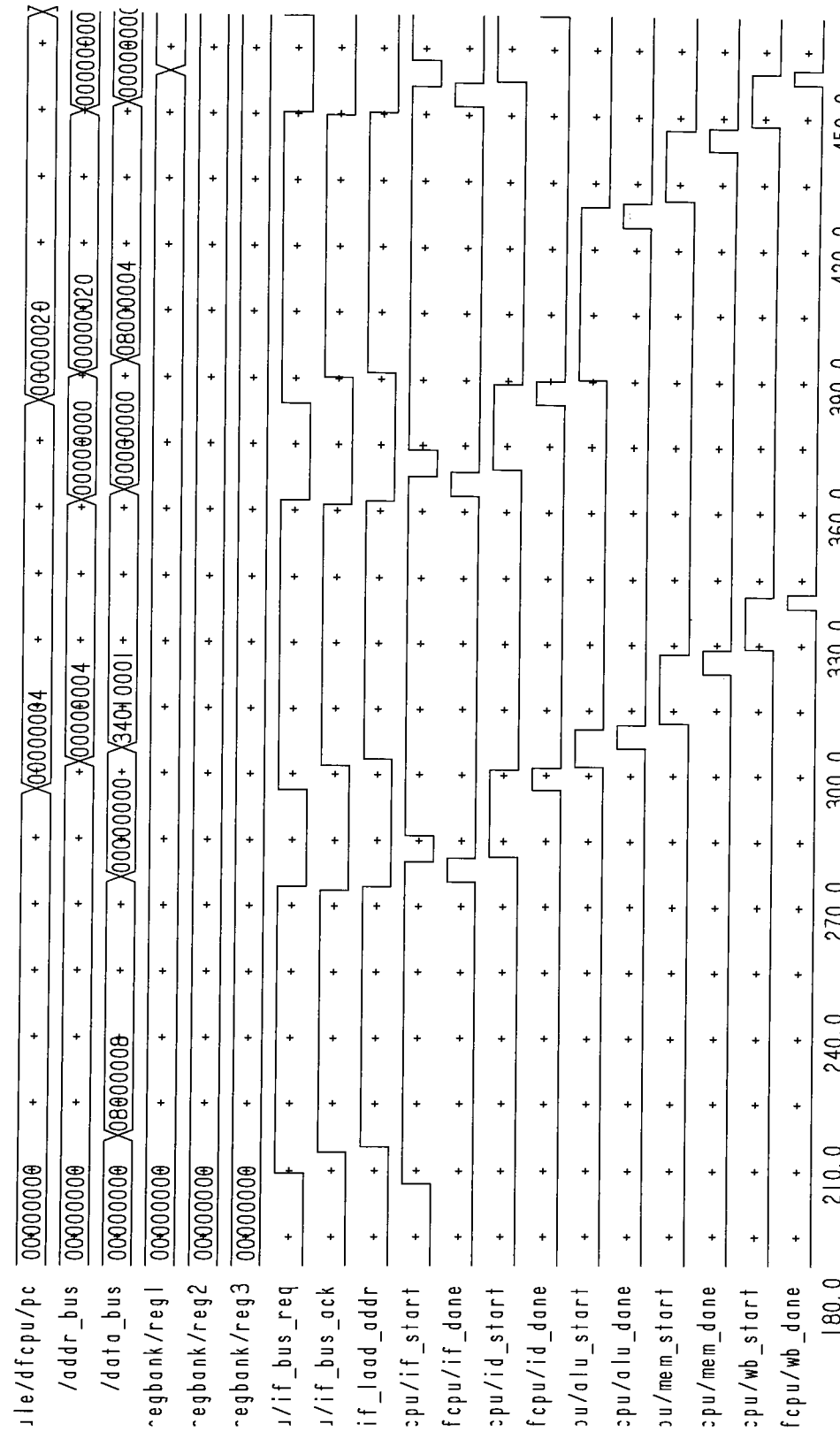


Figure 6-3. Example 3 - Jump Instruction Waveforms

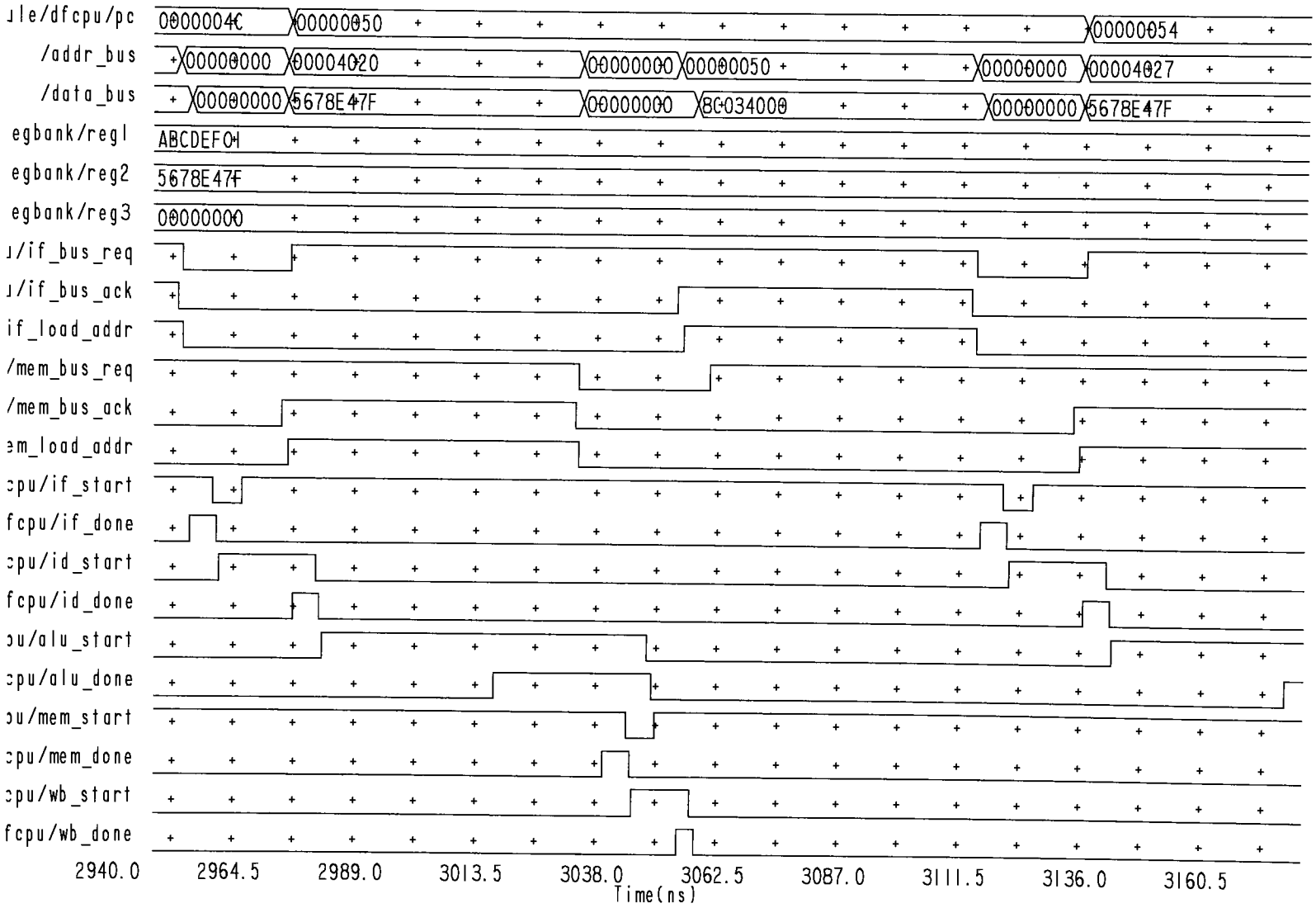


Figure 6-4. Example 4 - Load Instruction Waveforms

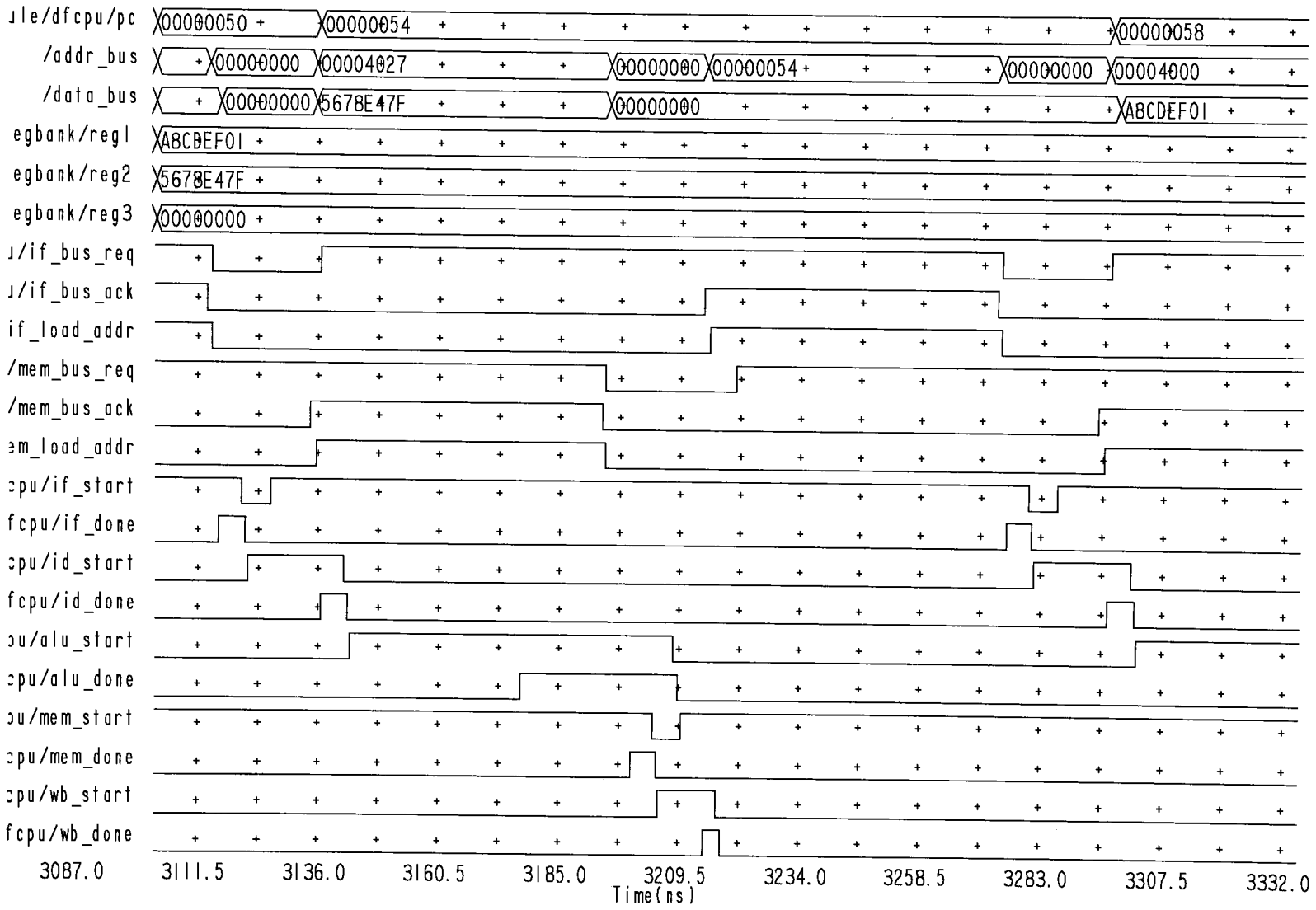


Figure 6-4 Continued

Figure 6-4 Continued

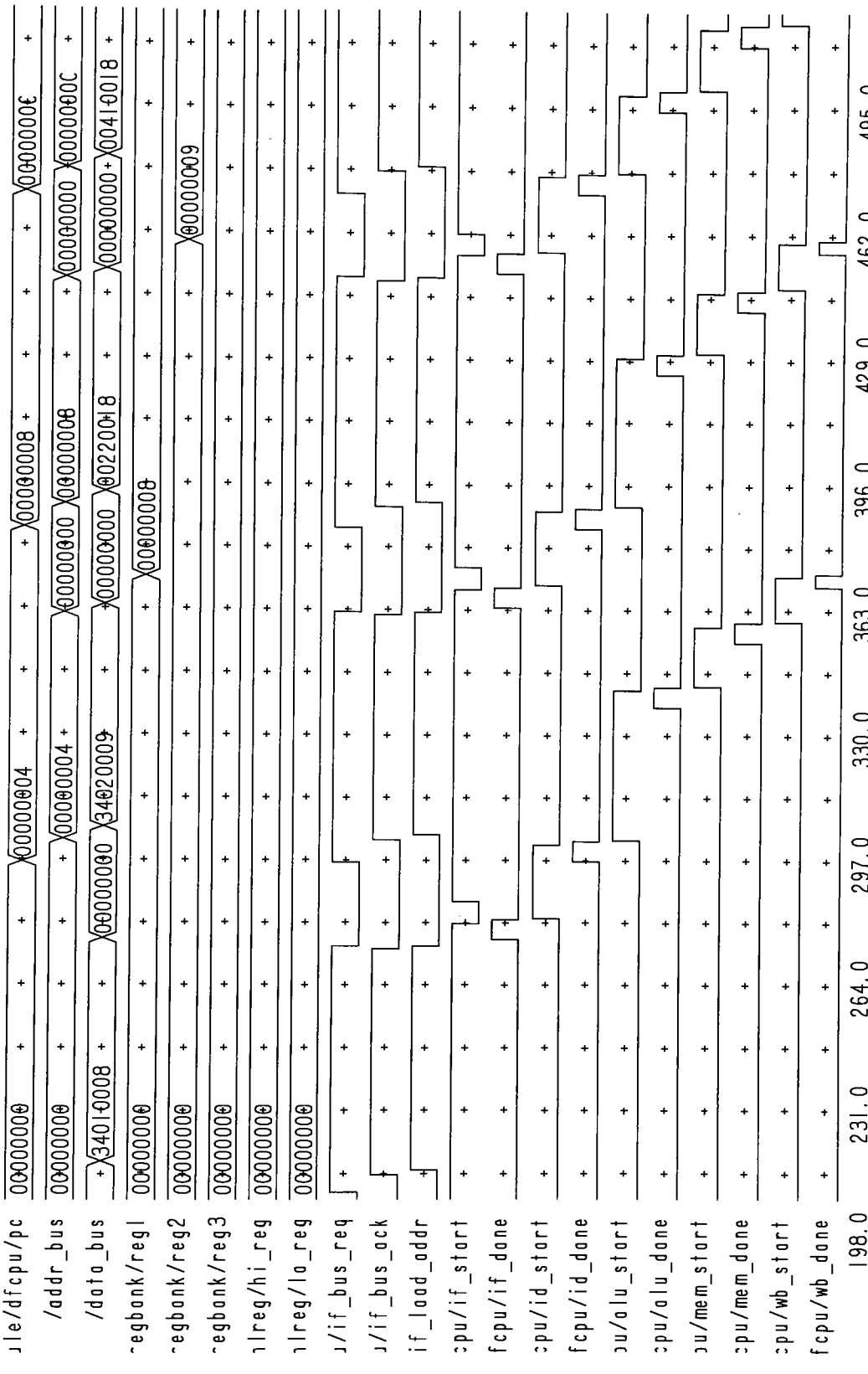


Figure 6-5. Example 5 - Multiplication Instruction Waveforms

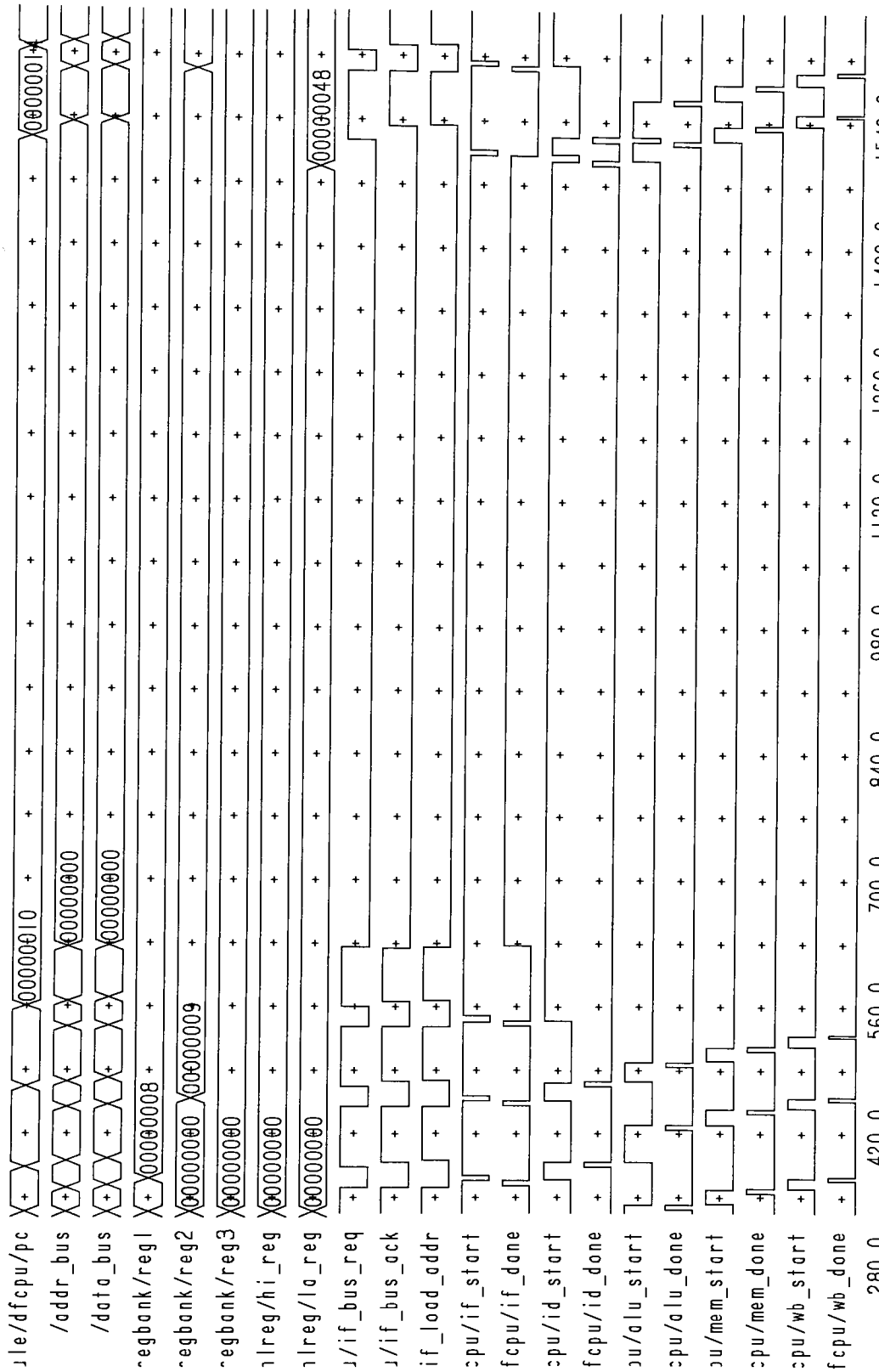


Figure 6-5 Continued

6.4 PROCESSOR SIMULATION TIMES

A program was written to calculate the processor execution time, the processor speed rating in millions of instructions per second (MIPS), and the actual execution time on the Quicksim. This program, shown in Figure 6-6, consists of a ten instruction loop. Register 1 (*r1*) is set to a value of 1000 (3e8 hex). The value is decremented with each iteration of the loop. The loop is exited when the value in *r1* equals zero.

```
# program to test simulation
# and actual run time

ori     r1    r0    64      # value = 100
ori     r9    r0    1
lui     r2    5555          # ***
ori     r3    r0    a
andi    r4    r2    abcd
sll     r5    r3    4
sw      r4    4000    0
add     r6    r3    r5
lw      r7    4000    0
sub     r1    r1    r9      # decrement value
bne     r1    r0    fff7    # branch to *** if value not 0
sll     r0    r0    0
break
```

Figure 6-6. Program to Calculate Processor Execution Times

The MIPS rating is calculated by dividing the number of instructions executed by the simulation time. The actual execution time (how long it took Quicksim to run this program) is calculated using a wrist watch and viewing the trace window. The data and calculations are shown below.

Number of instructions executed:	**1000**
Total processor execution time (ns):	**102,420**
MIPS = (1000 inst / 102420 ns) (10e9 ns / 1 second) =	**9.8**
Actual execution time (minutes):	**7.5**

7.0 CONCLUSIONS

This thesis has shown that an asynchronous design of a microprocessor is a viable alternative to synchronous design. It has also shown that a VHDL top down design methodology is an effective approach to modern circuit design. Conclusions can also be drawn from the results of the model testing.

The VHDL dataflow model achieved an average speed rating of 9.8 MIPS, as shown in section 6.4. Any comparisons to the synchronous version are difficult to make due mainly to two reasons. First, the actual speed of the asynchronous processor is variable and depends on which instructions are executed. Secondly, the memory was not implemented in the asynchronous version.

The testing procedure using the VHDL test bench is easy to use and provides a fast, accurate, and efficient means to test the processor. Back annotating delay times from Accusim circuit simulations proved an effective way to design. It was easy to modify the design by changing the delay times and recompiling the source code. If a problem occurred during execution then the code could be stepped through with the VHDL debugger. Lastly, the example simulations proved that the models worked and that the asynchronous processor was designed and working to specifications.

The first advantage of the asynchronous design methodology is the elimination of the global clock. In synchronous design, the global clock is routed to all areas of the chip. This is done to synchronize all the circuit modules. These metal clock lines become very long since they are routed from one side of the chip to the other. Due to the different lengths of metal lines and different capacitive loads, one area of the chip may receive the clock signal earlier or later than another area, creating clock skew. Clock skew becomes more of a problem with this type of design as circuits become more dense. This is not a problem in asynchronous design because the global clock is eliminated.

The second advantage of asynchronous design is that every module can operate at maximum speed. In synchronous design, the clock cycle time is dictated by the slowest module or the longest instruction. The synchronous processor has to be designed to handle the worst case operation. This causes most instructions to use only part of the clock cycle. This leads to long periods of idle time for the processor. This is not the case in asynchronous design. Since this type of design is based on events, when a stage or module is complete it signals the next module which reduces or eliminates idle time between modules. A module can go as fast as a previous module can send data and the next module can receive data.

The third advantage of asynchronous design is that every component is modular in construction and use. Start and completion signals are inherent in asynchronous design. A start signal is sent to the module to begin processing. When a module is finished it sends a completion signal to the next module. This allows every component or module to be replaced and/or redesigned at any time. The only requirement is that input and output signals match (i.e. the "black box" approach). This modular concept is most helpful for specifying memory requirements. Memory of any speed can be used in asynchronous design. The memory interface will wait until the memory sends an acknowledgment that it is finished.

One disadvantage of asynchronous design is that events cannot be predicted. Asynchronous design is well suited for system designs that only require connections between adjacent modules. However, multi-connection modules are more difficult to coordinate and circuit complexity is increased. An example that makes this apparent is a *branch* instruction. Three stages of the pipeline (IF, ID, and ALU) need to be coordinated to execute a *branch* instruction. The IF stage has to know which instruction to execute next (i.e. whether or not the branch is taken). The ID stage has to calculate the branch destination address. The ALU stage determines whether or not the branch is taken. The synchronous version implements these tasks by using a two-phase clock. The IF can wait

until the second phase of the clock to perform an instruction fetch. This allows the ID to determine the state of the branch. However, in the asynchronous version, the IF has to stall until the other stages have done their tasks. Since every action is event based and not time based, it is not known when these tasks will start and finish. Controlling circuitry has to be added to the design to enforce an order of operations.

Another disadvantage of asynchronous design occurs when a completion signal is generated. The two methods used are dummy delay and differential cascade voltage switch logic (DCVSL). These issues are discussed in Kevin Johnson's thesis [10] and Scott Siers' thesis [19].

The flexibility of VHDL allows a seamless transition through different levels of abstraction. VHDL provides the designer a way of writing different model abstractions with the same familiar constructs. The high level constructs allow describing the behavior or function of a system and is very much like a typical programming language. It also provides a means of writing a test bench. The low level constructs allow the structure or gate level of a system to be modeled with precise timing information. If no particular type of model is adequate, VHDL allows the designer to mix different types of modeling styles.

Top down design provides the designer with a method to guide a design through all levels of abstraction. A behavioral model, which projects a high level of abstraction, frees the designer of implementation details and allows for concentration on the basic system functionality. Once the behavior is defined, then more precise models can be written describing lower level details. As timing information is annotated into the models, the models are verified for correct operation. A test bench provides a fast, accurate, and efficient means to accomplish verification. This constant switching between designing and testing allows the designer to progress at a reasonable fashion without going too far with a bad design. The designer is alerted to a possible design flaw in the early stages of design. At this point the designer could even go back to the behavioral model and reassess its

functionality. The actual board or component is not produced until the design process is complete, which saves time and money.

One general problem with this thesis was the decision to take a synchronous processor and convert it to an asynchronous design. Some of the instructions in the R3000's instruction set were not well suited to an asynchronous approach. As an example consider the two branch and link instructions. These instructions were very complicated because they have to do all the tasks associated with a branch (branch address calculation and branch determination) and also store a return address in the link register. Since both adders are busy (the AA is busy calculating the branch address and the ALUC determines if the branch is taken), another adder (add8) was designed to handle the link address calculation. All these tasks are done in parallel to prevent the pipeline from stalling. It would have been better to design a processor to match the strengths of asynchronous design.

The requirement that each thesis be an independent work was a major disadvantage to the concept of a team project. Unfortunately, the team members divided the project in a manner which created separate pieces that were difficult to work on concurrently. This ultimately limited the effectiveness of the team concept. All VHDL modeling should have been done before any circuit design and layout was attempted. Throughout the entire design cycle, the VHDL modeling and simulation caught many design errors that were not apparent at first to the individual designers. These design errors were substantial enough to force complete redesigns which cost the designers a great deal of time and effort.

Overall, this thesis was a tremendous learning experience. The team concept provided the vehicle to tackle this large project. This thesis has shown the benefits of asynchronous and top down design methodologies. VHDL has proved to be an indispensable tool for design and development of an electronic system.

BIBLIOGRAPHY

[1] Asada, Kunihiro, Okura Kazama, and Kyoung-rok Cho, "Design of a Self-timed Data Path for a Fully Asynchronous Microprocessor", International Workshop SASIMI'92, Kobe, Japan, 1992.

[2] Ashenden, Peter J., "The VHDL Cookbook", First Edition, Department of Computer Science, University of Adelaide, South Australia, July 1990.

[3] Bhasker, Jayaram, A VHDL Primer, Prentice-Hall, New Jersey, 1992.

[4] Dewilde, Prof. Dr. Ir. P. M., Dr. Ir. R. Nouta, and Ir. M. Sim, " A Self-Timed Handshake-Scheme Applied to a Bit-Serial Multiplier", Technical Report 91-107, Delft University of Technology, June 1991.

[5] Ginosar, Ran, and Nick Mitchell, "On the Potential of Asynchronous Pipelined Processors", *Computer Architecture News*, Vol. 18, No. 4, December 1990.

[6] Goeke, James A., "Design of a Hardware Efficient Key Generation Algorithm with a VHDL Implementation", Rochester Institute of Technology, May 1993.

[7] Hayes, John P., Computer Architecture and Organization, McGraw-Hill, New York, 1988.

[8] Ilana, David, Ran Ginosar, Michael Yeoli, "An Efficient Implementation of Boolean Functions as Self-Timed Circuits", *IEEE Transactions on Computers*, Vol. 41, No. 1, January 1992.

[9] Jacobs, Gordon M., and Robert Brodersen, "A Fully Asynchronous Digital Signal Processor Using Self-Timed Circuits", *IEEE Journal of Solid-State Circuits*, Vol. 25, No. 6, December 1990.

[10] Johnson, Kevin C., "Design and Implementation of an Asynchronous Version of the MIPS R3000 Microprocessor", Rochester Institute of Technology, January 1994.

[11] Kane, Gerry, MIPS RISC Architecture, Prentice-Hall, New Jersey, 1988.

[12] Kane, Gerry, and Joe Heinrich, MIPS RISC Architecture, Prentice-Hall, New Jersey, 1992.

[13] Mano, M. Morris, Computer Engineering Hardware Design, Prentice-Hall, New Jersey, 1988.

[14] McAuley, Anthony J., "Four State Asynchronous Architectures", *IEEE Transactions on Computers*, Vol. 41, No. 2, February 1992.

[15] McCluskey, Edward J., Logic Design Principles, Prentice-Hall, New Jersey, 1986.

[16] Mukherjee, Amar, Introduction to nMOS and CMOS VLSI Systems Design, Prentice-Hall, New Jersey, 1986.

[17] Navabi, Zainalabedin, VHDL Analysis and Modeling of Digital Systems, McGraw-Hill, New York, 1993.

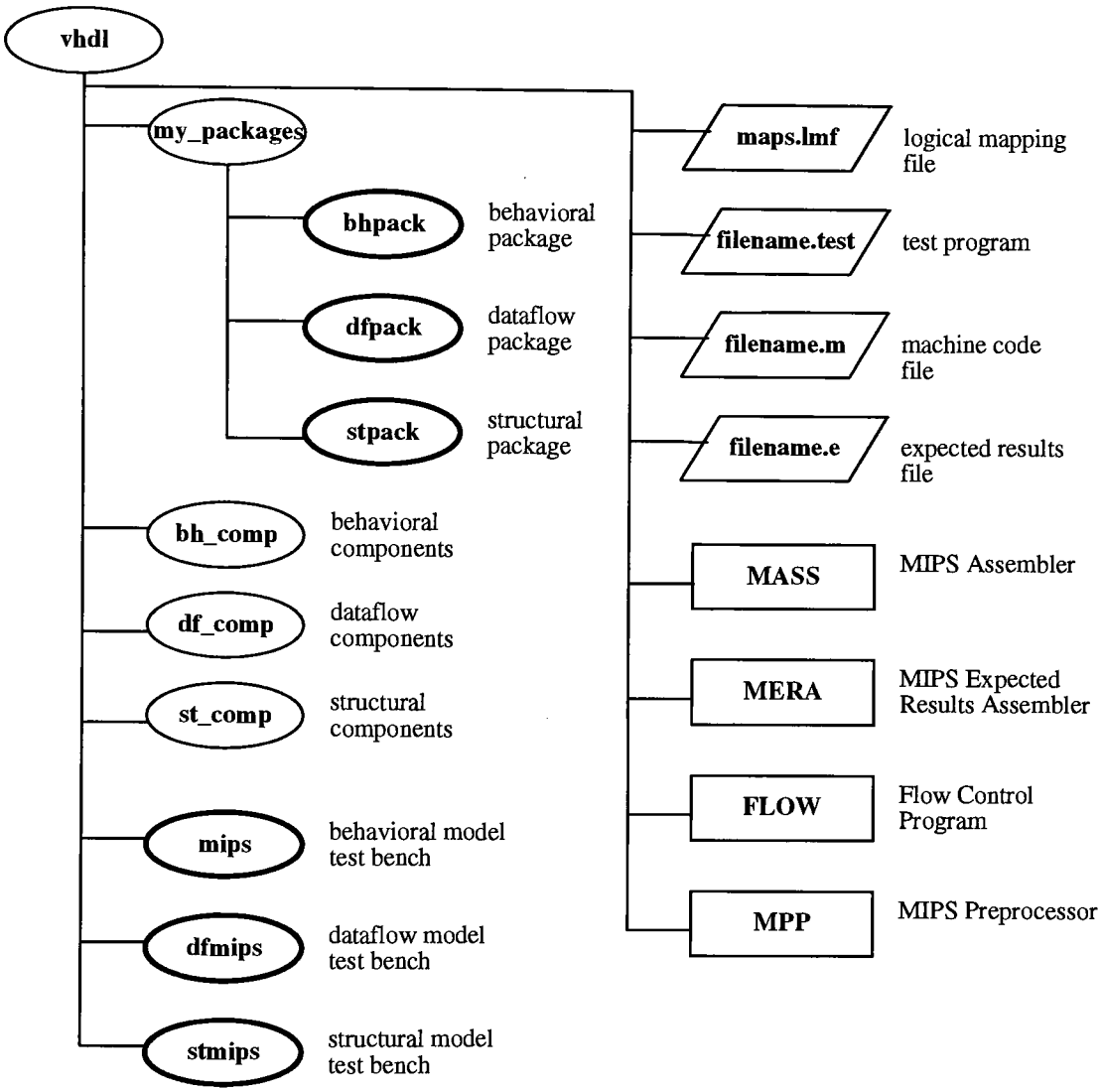
[18] Nouta, R., M. Sim, and Gunawan, "Two Self-Timed Handshake Controllers for High Speed Applications", IEEE, 1992.

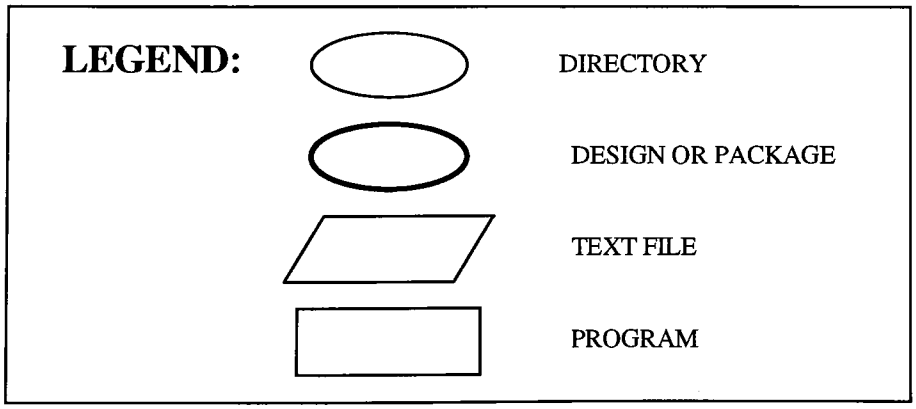
[19] Siers, Scott E., "Design and Implementation of an Asynchronous Version of the MIPS R3000 Microprocessor", Rochester Institute of Technology, November 1993.

[20] Seitz, Charles I., "Self-Timed VLSI Systems", CalTech Conference on VLSI, January 1979.

[21] Tan, Y. K., and Y. C. Lim, "Self-Timed System Design Technique", *Electronic Letters*, Vol. 26, No. 5, March 1990.

APPENDIX A - FILE STRUCTURE





APPENDIX B - BEHAVIORAL MODEL SOURCE CODE

COMPONENT:	MIPS
FILENAME:	MIPS_E.VHDL
DESCRIPTION:	Test bench for behavioral model of asynchronous version of MIPS R3000 microprocessor (entity)

```
-- library and use clauses
library mgc_portable, ieee;
use mgc_portable.qsim_logic.all;
use mgc_portable.qsim_relations.all;
use ieee.std_logic_1164.all;

--use std.textio.all;

library my_packages;
use my_packages.package_1.all;

library my_components;
use my_components.cpu.all;
use my_components.memory.all;
use my_components.compare.all;

entity mips is
end mips;
```

COMPONENT:	MIPS
FILENAME:	MIPS_A.VHDL
DESCRIPTION:	Test bench for behavioral model of asynchronous version of MIPS R3000 microprocessor (architecture)

```
architecture mips_a of mips is

    component memory
        port(mem_control_sig:   in    mem_control_type;
             addr_bus:          in    bit_30;
             addr_bus_lo:       in    bit_2;
             mem_ack_sig:       out   question_type;
             mem_exception_sig: out   question_type;
             data_bus:          inout bus_bit_32 bus);
    end component;

    component cpu
        port(sys_control_sig:     in
sys_control_type;
             mem_ack_sig:         in    question_type;
             mem_exception_sig:   in    question_type;
             mem_control_sig:     out
mem_control_type;
             addr_bus:            out   bit_30;
             addr_bus_lo:         out   bit_2;
             data_bus:            inout bus_bit_32 bus;
             compare_control_sig: out
compare_control_type;
             pc:                  out   bit_32;
             r1:                  out   bit_32;
             r2:                  out   bit_32;
             r3:                  out   bit_32;
             r4:                  out   bit_32;
             r31:                 out   bit_32;
             hi:                  out   bit_32;
             lo:                  out   bit_32;
             epc:                 out   bit_32;
             cause:               out   bit_32;
             compare_ack_sig:     in    question_type);
    end component;

    component compare
        port(compare_control_sig: in
compare_control_type;
             pc:                  in    bit_32;
             r1:                  in    bit_32;
             r2:                  in    bit_32;
             r3:                  in    bit_32;
             r4:                  in    bit_32;
             r31:                 in    bit_32;
             hi:                  in    bit_32;
             lo:                  in    bit_32;
             epc:                 in    bit_32;
             cause:               in    bit_32;
             compare_ack_sig:     out   question_type);
    end component;

    signal sys_control_sig:     sys_control_type;
    signal mem_control_sig:     mem_control_type;
    signal mem_ack_sig:         question_type;
    signal mem_exception_sig:   question_type;
    signal addr_bus:            bit_30;
    signal addr_bus_lo:         bit_2;
    signal data_bus:            bus_bit_32 bus;
    signal compare_control_sig: compare_control_type;
    signal pc:                  bit_32;
    signal r1:                  bit_32;
    signal r2:                  bit_32;
    signal r3:                  bit_32;
    signal r4:                  bit_32;
    signal r31:                 bit_32;
    signal hi:                  bit_32;
    signal lo:                  bit_32;
    signal epc:                 bit_32;
    signal cause:               bit_32;
    signal compare_ack_sig:     question_type;

begin

    mem: memory
        port map(mem_control_sig, addr_bus,
addr_bus_lo,
                 mem_ack_sig, mem_exception_sig,
data_bus);

    proc: cpu
        port map(sys_control_sig, mem_ack_sig,
mem_exception_sig,
                 mem_control_sig, addr_bus,
addr_bus_lo, data_bus,
                 compare_control_sig, pc, r1, r2, r3,
r4, r31, hi,
                 lo, epc, cause, compare_ack_sig);

    comp: compare
        port map(compare_control_sig, pc, r1, r2, r3,
r4, r31, hi,
                 lo, epc, cause, compare_ack_sig);

end mips_a;
```

COMPONENT:	CPU
FILENAME:	CPU_E.VHDL
DESCRIPTION:	Central Processing Unit (CPU) component of behavioral model test bench (entity)

```
-- library and use clauses
library mgc_portable, ieee;
use mgc_portable.qsim_logic.all;
use mgc_portable.qsim_relations.all;
use ieee.std_logic_1164.all;

use std.textio.all;

library my_packages;
use my_packages.package_1.all;

entity cpu is
    port(sys_control_sig:     in    sys_control_type;
         mem_ack_sig:         in    question_type;
         mem_exception_sig:   in    question_type;
         mem_control_sig:     out   mem_control_type;
         addr_bus:            out   bit_30;
         addr_bus_lo:         out   bit_2;
         data_bus:            inout bus_bit_32 bus;
         compare_control_sig: out
compare_control_type;
         pc:                  out   bit_32;
         r1:                  out   bit_32;
         r2:                  out   bit_32;
         r3:                  out   bit_32;
```

```
        r4:                 out   bit_32;
        r31:                out   bit_32;
        hi:                 out   bit_32;
        lo:                 out   bit_32;
        epc:                out   bit_32;
        cause:              out   bit_32;
        compare_ack_sig:    in    question_type);
end cpu;
```

COMPONENT:	**CPU**
FILENAME:	**CPU_A.VHDL**
DESCRIPTION:	**Central Processing Unit (CPU) component of behavioral model test bench (architecture)**

```
architecture cpu_a of cpu is

    ------------------------------------------------
--------------------
    -- signals for processor
    ------------------------------------------------
--------------------
    signal inst: inst_type;   -- instruction mneumonic

    --signal pc:  bit_32;        -- program counter

    signal r0:  bit_32;        -- registers r0 thru r31
    --signal r1:  bit_32;
    --signal r2:  bit_32;
    --signal r3:  bit_32;
    --signal r4:  bit_32;
    signal r5:  bit_32;
    signal r6:  bit_32;
    signal r7:  bit_32;
    signal r8:  bit_32;
    signal r9:  bit_32;
    signal r10: bit_32;
    signal r11: bit_32;
    signal r12: bit_32;
    signal r13: bit_32;
    signal r14: bit_32;
    signal r15: bit_32;
    signal r16: bit_32;
    signal r17: bit_32;
    signal r18: bit_32;
    signal r19: bit_32;
    signal r20: bit_32;
    signal r21: bit_32;
    signal r22: bit_32;
    signal r23: bit_32;
    signal r24: bit_32;
    signal r25: bit_32;
    signal r26: bit_32;
    signal r27: bit_32;
    signal r28: bit_32;
    signal r29: bit_32;
    signal r30: bit_32;
    --signal r31: bit_32;

    --signal hi:  bit_32;          -- hi, lo registers
used for mult and div
    --signal lo:  bit_32;

    signal exception_sig:     exception_type;
    --signal epc:   bit_32;       -- exception handling
    --signal cause: bit_32;


begin

    processor: process

        ------------------------------------------------
--------------------
        --variables for control
        ------------------------------------------------
--------------------
        variable run_mode_flag:   question_type;
        variable delay_slot_flag: delay_slot_type;
        variable latency_flag:    delay_slot_type;
        variable left_flag:       delay_slot_type;
        variable right_flag:      delay_slot_type;
        variable exception_flag:  exception_type;
        variable bd_flag:         question_type;

        ------------------------------------------------
--------------------
        --variables for storage
        ------------------------------------------------
--------------------
        variable pc_reg:  bit_32;
        variable pc_temp: bit_32;
        variable hi_reg:  bit_32;
        variable lo_reg:  bit_32;
        type reg_array is array (bit_5_range) of
bit_32;
        variable reg: reg_array;

        ------------------------------------------------
--------------------
        --variables for decoding instructions
        ------------------------------------------------
--------------------
        variable current_inst: bit_32;
        variable opcode:       bit_6;
        variable funct:        bit_6;
        variable opcode_seg:   bit_3;
        variable code:         bit_20;
        variable rs:           bit_5_range;
        variable rt:           bit_5_range;
        variable rd:           bit_5_range;
        variable shamt:        bit_5_range;
        variable base:         bit_5_range;
        variable immed:        bit_16;
        variable offset:       bit_16;
        variable target:       bit_26;
        variable reg_funct:    bit_5;

        ------------------------------------------------
--------------------
        --temporary working variables
        ------------------------------------------------
--------------------
        variable ea:              bit_32;
        variable amt:             bit_5_range;
        variable mult_temp:       bit_64;
        variable ui:              bit_vector(31 downto
16);
        variable temp:            bit_32;
        variable temp_reg_val_1: bit_32;
        variable temp_reg_val_2: bit_32;
        variable temp_reg_num_1: bit_5_range;
        variable temp_reg_num_2: bit_5_range;
        variable temp_addr_1:     bit_2;
        variable temp_addr_2:     bit_2;

        ------------------------------------------------
--------------------
        --variables for exception handling
        ------------------------------------------------
--------------------
        variable epc_reg:   bit_32;
        variable cause_reg: bit_32;
        variable ovrflw:    bit;


        ------------------------------------------------
--------------------
        --procedures...
        ------------------------------------------------
--------------------
        procedure mem_read(addr: in bit_32; size: in
size_type;
                           result: out bit_32) is
        begin
            addr_bus <= addr(31 downto 2) after delay;
            addr_bus_lo <= addr(1 downto 0) after
delay;
            case size is
              when byte =>
                mem_control_sig <= read_b after delay;
              when ubyte =>
                mem_control_sig <= read_ub after delay;
              when halfword =>
                mem_control_sig <= read_hw after delay;
              when uhalfword =>
                mem_control_sig <= read_uhw after
delay;
              when word =>
                mem_control_sig <= read_w after delay;
              when lefty =>
                mem_control_sig <= read_l after delay;
              when righty =>
                mem_control_sig <= read_r after delay;
            end case;
            wait until mem_ack_sig = yes;
            result := data_bus;
            mem_control_sig <= reset after delay;
            wait until mem_ack_sig = no;
            if mem_exception_sig = yes then
                exception_flag := addr_load;
```

```
            end if;
        end mem_read;



        procedure mem_write(addr: in bit_32; size: in
size_type;
                            data: in bit_32) is
        begin
            addr_bus <= addr(31 downto 2) after delay;
            addr_bus_lo <= addr(1 downto 0) after
delay;
            data_bus <= data after delay;
            case size is
              when byte =>
                  mem_control_sig <= write_b after
delay;
              when halfword =>
                  mem_control_sig <= write_hw after
delay;
              when word =>
                  mem_control_sig <= write_w after
delay;
              when lefty =>
                  mem_control_sig <= write_l after
delay;
              when righty =>
                  mem_control_sig <= write_r after
delay;
              when others =>
                  null;
            end case;
            wait until mem_ack_sig = yes;
            mem_control_sig <= reset after delay;
            wait until mem_ack_sig = no;
            data_bus <= null after delay;
            if mem_exception_sig = yes then
                exception_flag := addr_store;
            end if;
        end mem_write;


    begin
          wait on sys_control_sig;      -- entry into
processor process

          case sys_control_sig is

            ---------------------------------------------
--------------------
            -- stop processor (variable used to stop
process immediately)
            ---------------------------------------------
--------------------
            when stop =>
              run_mode_flag := no;

            ---------------------------------------------
--------------------
            -- reset signals and variables
            ---------------------------------------------
--------------------
            when reset =>
              mem_control_sig <= reset;       --
control signals
              compare_control_sig <= reset;

              run_mode_flag     := no;          --
control variables
              delay_slot_flag := ignore;
              latency_flag    := ignore;
              left_flag       := ignore;
              right_flag      := ignore;
              exception_flag  := idle;
              bd_flag         := no;

              data_bus    <= null;            -- data,
address signals
              inst        <= nop;
              addr_bus    <=
b"00_0000_0000_0000_0000_0000_0000_0000";
              addr_bus_lo <= b"00";
              pc          <= x"0000_0000",

              pc_reg      := x"0000_0000";    --
storage variables
              reg(0)      := x"0000_0000";
              reg(1)      := x"0000_0000";
              reg(2)      := x"0000_0000";
              reg(3)      := x"0000_0000";
              reg(4)      := x"0000_0000";
              reg(5)      := x"0000_0000";
              reg(6)      := x"0000_0000";
              reg(7)      := x"0000_0000";
              reg(8)      := x"0000_0000";
              reg(9)      := x"0000_0000";
              reg(10)      := x"0000_0000";
              reg(11)      := x"0000_0000";
              reg(12)      := x"0000_0000";
              reg(13)      := x"0000_0000";
              reg(14)      := x"0000_0000";
              reg(15)      := x"0000_0000";
              reg(16)      := x"0000_0000";
              reg(17)      := x"0000_0000";
              reg(18)      := x"0000_0000";
              reg(19)      := x"0000_0000";
              reg(20)      := x"0000_0000";
              reg(21)      := x"0000_0000";
              reg(22)      := x"0000_0000";
              reg(23)      := x"0000_0000";
              reg(24)      := x"0000_0000";
              reg(25)      := x"0000_0000";
              reg(26)      := x"0000_0000";
              reg(27)      := x"0000_0000";
              reg(28)      := x"0000_0000";
              reg(29)      := x"0000_0000";
              reg(30)      := x"0000_0000";
              reg(31)      := x"0000_0000";
              epc_reg     := x"0000_0000";
              cause_reg   := x"0000_0000";

            ---------------------------------------------
--------------------
            -- load program into memory
            ---------------------------------------------
--------------------
            when load =>
              run_mode_flag := no;

              mem_control_sig <= load after delay;  --
memory handshake
              wait until mem_ack_sig = yes;
              mem_control_sig <= reset after delay;
              wait until mem_ack_sig = no;

              compare_control_sig <= load after delay;
-- compare handshake
              wait until compare_ack_sig = yes;
              compare_control_sig <= reset after delay;
              wait until compare_ack_sig = no;

            ---------------------------------------------
--------------------
            -- run processor
            ---------------------------------------------
--------------------
            when run =>
              run_mode_flag := yes;
              while run_mode_flag = yes loop

                  -- fetch next instruction
                  mem_read(pc_reg, word, current_inst);

                  -- decode opcode of instruction
                  -- determine which type of addressing
it is
                  -- set fields accordingly

                  opcode := current_inst(31 downto 26);
                  opcode_seg := opcode(5 downto 3);


                  ---------------------------------------
----------------------
                  -- special instructions
                  ---------------------------------------
----------------------
                  if opcode = i_special then
                      -- use register instruction
format
                      -- (special, rs, rt, rd, shamt,
func)
                      -- except for break and syscall
instruction

                      funct := current_inst(5 downto
0);
                      rs := bton(current_inst(25 downto
21));
                      rt := bton(current_inst(20 downto
16));
                      rd := bton(current_inst(15 downto
11));
                      shamt := bton(current_inst(10
downto 6));
```

```
                    case funct is
                      when i_syscall =>
                        -- (special, 0    , syscall)
                        inst <= op_syscall;
                        exception_flag :=
syscall_trap;

                      when i_break =>
                        -- (special, code, break)
                        inst <= op_break;
                        --code := current_inst(25
downto 6);
                        exception_flag :=
breakpt_trap;

                      when i_sll =>
                        inst <= op_sll;
                        if rd /= 0 then
                            reg(rd) :=
shift_ll(reg(rt), shamt);
                        end if;

                      when i_srl =>
                        inst <= op_srl;
                        if rd /= 0 then
                            reg(rd) :=
shift_rl(reg(rt), shamt);
                        end if;

                      when i_sra =>
                        inst <= op_sra;
                        if rd /= 0 then
                            reg(rd) :=
shift_ra(reg(rt), shamt);
                        end if;

                      when i_sllv =>
                        inst <= op_sllv;
                        amt := bton(reg(rs) and
x"0000_001f");
                        if rd /= 0 then
                            reg(rd) :=
shift_ll(reg(rt), amt);
                        end if;

                      when i_srlv =>
                        inst <= op_srlv;
                        emt := bton(reg(rs) and
x"0000_001f");
                        if rd /= 0 then
                            reg(rd) :=
shift_rl(reg(rt), amt);
                        end if;

                      when i_srav =>
                        inst <= op_srav;
                        amt := bton(reg(rs) and
x"0000_001f");
                        if rd /= 0 then
                            reg(rd) :=
shift_ra(reg(rt), amt);
                        end if;

                      when i_jr =>
                        -- need one instruction delay
!!!
                        inst <= op_jr;
                        pc_temp := reg(rs);
                        delay_slot_flag := set;

                      when i_jalr =>
                        -- need one instruction delay
!!!
                        inst <= op_jalr;
                        pc_temp := reg(rs);
                        -- the address of the inst
after the delay slot
                        -- is placed in rd
                        if rd /= 0 then
                            reg(rd) := pc_reg +
x"0000_0008";
                        end if;
                        delay_slot_flag := set;

                      when i_mfhi =>
                        inst <= op_mfhi;
                        if rd /= 0 then
                            reg(rd) := hi_reg;
                        end if;

                      when i_mthi =>
                        inst <= op_mthi;
                        hi_reg := reg(rs);

                      when i_mflo =>
                        inst <= op_mflo;
                        if rd /= 0 then
                            reg(rd) := lo_reg;
                        end if;

                      when i_mtlo =>
                        inst <= op_mtlo;
                        lo_reg := reg(rs);

                      when i_mult =>
                        inst <= op_mult;
                        mult_temp := mult(reg(rs),
reg(rt));
                        lo_reg := mult_temp(31 downto
0);
                        hi_reg := mult_temp(63 downto
32);

                      when i_multu =>
                        inst <= op_multu;
                        mult_temp := reg(rs) *
reg(rt);
                        lo_reg := mult_temp(31 downto
0);
                        hi_reg := mult_temp(63 downto
32);

                      when i_div =>
                        inst <= op_div;
                        mult_temp := div(reg(rs),
reg(rt));
                        lo_reg := mult_temp(31 downto
0);
                        hi_reg := mult_temp(63 downto
32);

                      when i_divu =>
                        inst <= op_divu;
                        mult_temp := reg(rs) /
reg(rt);
                        lo_reg := mult_temp(31 downto
0);
                        hi_reg := mult_temp(63 downto
32);

                      when i_add =>
                        inst <= op_add;
                        if rd /= 0 then
                            add_ovf(reg(rs), reg(rt),
reg(rd), ovrflw);
                            if ovrflw = '1' then
                                exception_flag :=
overflow;
                            end if;
                        end if;

                      when i_addu =>
                        -- same as add except never
causes ovrflw exception
                        inst <= op_addu;
                        if rd /= 0 then
                            reg(rd) := reg(rs) +
reg(rt);
                        end if;

                      when i_sub =>
                        inst <= op_sub;
                        if rd /= 0 then
                            sub_ovf(reg(rs), reg(rt),
reg(rd), ovrflw);
                            if ovrflw = '1' then
                                exception_flag :=
overflow;
                            end if;
                        end if;

                      when i_subu =>
                        -- same as sub except never
causes ovrflw exception
                        inst <= op_subu;
                        if rd /= 0 then
                            reg(rd) := reg(rs) -
reg(rt);
                        end if;

                      when i_and =>
                        inst <= op_and;
                        if rd /= 0 then
                            reg(rd) := reg(rs) and
reg(rt);
                        end if;
```

```
                        when i_or =>
                          inst <= op_or;
                          if rd /= 0 then
                              reg(rd) := reg(rs) or
reg(rt);
                          end if;

                        when i_xor =>
                          inst <= op_xor;
                          if rd /= 0 then
                              reg(rd) := reg(rs) xor
reg(rt);
                          end if;

                        when i_nor =>
                          inst <= op_nor;
                          if rd /= 0 then
                              reg(rd) := reg(rs) nor
reg(rt);
                          end if;

                        when i_slt =>
                          inst <= op_slt;
                          if rd /= 0 then
                              if btoi(reg(rs)) <
btoi(reg(rt)) then
                                  reg(rd) :=
x"0000_0001";
                              else
                                  reg(rd) :=
x"0000_0000";
                              end if;
                          end if;

                        when i_sltu =>
                          inst <= op_sltu;
                          if rd /= 0 then
                              if bton(reg(rs)) <
bton(reg(rt)) then
                                  reg(rd) :=
x"0000_0001";
                              else
                                  reg(rd) :=
x"0000_0000";
                              end if;
                          end if;

                        when others =>
                          -- error,reserved instruction
                          inst <= reserved;
                          exception_flag :=
reserved_inst;

                    end case;


                    ------------------------------------
-----------------------
                    -- conditional branch (bcond)
instructions
                    ------------------------------------
-----------------------
                    elsif opcode = i_bcond then
                        -- use immediate instruction
format
                        -- function is found at rt field
                        -- (bcond, rs, func, offset)

                        rs := bton(current_inst(25 downto
21));
                        reg_funct := current_inst(20
downto 16);
                        offset := current_inst(15 downto
0);

                        case reg_funct is
                          when i_bltz =>
                            inst <= op_bltz;
                            if reg(rs)(31) = '1' then
                                pc_temp := pc_reg +
x"0000_0004";
                                pc_temp := pc_temp +
shift_ll(se16to32(offset),2);
                                delay_slot_flag := set;
                            end if;

                          when i_bgez =>
                            inst <= op_bgez;
                            if reg(rs)(31) = '0' then
                                pc_temp := pc_reg +
x"0000_0004";
                                pc_temp := pc_temp +
shift_ll(se16to32(offset),2);
                                delay_slot_flag := set;
                            end if;

                          when i_bltzal =>
                            inst <= op_bltzal;
                            reg(31) := pc_reg +
x"0000_0008";
                            if reg(rs)(31) = '1' then
                                pc_temp := pc_reg +
x"0000_0004";
                                pc_temp := pc_temp +
shift_ll(se16to32(offset),2);
                                delay_slot_flag := set;
                            end if;

                          when i_bgezal =>
                            inst <= op_bgezal;
                            reg(31) := pc_reg +
x"0000_0008";
                            if reg(rs)(31) = '0' then
                                pc_temp := pc_reg +
x"0000_0004";
                                pc_temp := pc_temp +
shift_ll(se16to32(offset),2);
                                delay_slot_flag := set;
                            end if;

                          when others =>
                            -- error, reserved
instruction
                            inst <= reserved;
                            exception_flag :=
reserved_inst;

                        end case;

                    ------------------------------------
-----------------------
                    -- jump, jump and link (using target)
instructions
                    ------------------------------------
-----------------------
                    elsif opcode = i_j or opcode = i_jal
then
                        -- use jump instruction format
                        -- (op, target)

                        target := current_inst(25 downto
0);

                        case opcode is
                          when i_j =>
                            inst <= op_j;
                            pc_temp := pc_reg(31 downto
28) & target & b"00";
                            delay_slot_flag := set;

                          when i_jal =>
                            inst <= op_jal;
                            pc_temp := pc_reg(31 downto
28) & target & b"00";
                            reg(31) := pc_reg +
x"0000_0008";
                            delay_slot_flag := set;

                          when others =>
                            null;

                        end case;

                    ------------------------------------
-----------------------
                    -- branch instructions
                    ------------------------------------
-----------------------
                    elsif opcode_seg = b"000" then
                        -- for i_beq, i_bne, i_blez,
i_bgtz
                        -- use immediate instruction
format
                        -- (op, rs, rt, offset)

                        rs := bton(current_inst(25 downto
21));
                        rt := bton(current_inst(20 downto
16));
                        offset := current_inst(15 downto
0);

                        case opcode is
                          when i_beq =>
```

```
                        inst <= op_beq;
                        if reg(rs) = reg(rt) then
                            pc_temp := pc_reg +
x"0000_0004";
                            pc_temp := pc_temp +
shift_ll(se16to32(offset),2);
                            delay_slot_flag := set;
                        end if;

                      when i_bne =>
                        inst <= op_bne;
                        if reg(rs) /= reg(rt) then
                            pc_temp := pc_reg +
x"0000_0004";
                            pc_temp := pc_temp +
shift_ll(se16to32(offset),2);
                            delay_slot_flag := set;
                        end if;

                      when i_blez =>
                        inst <= op_blez;
                        if reg(rs)(31) = '1' or
reg(rs) = reg(0) then
                            pc_temp := pc_reg +
x"0000_0004";
                            pc_temp := pc_temp +
shift_ll(se16to32(offset),2);
                            delay_slot_flag := set;
                        end if;

                      when i_bgtz =>
                        inst <= op_bgtz;
                        if reg(rs)(31) = '0' and
reg(rs) /= reg(0) then
                            pc_temp := pc_reg +
x"0000_0004";
                            pc_temp := pc_temp +
shift_ll(se16to32(offset), 2);
                            delay_slot_flag := set;
                        end if;

                      when others =>
                        null;

                    end case;


                    ------------------------------------
-----------------------
                    -- ALU immediate instructions
                    ------------------------------------
-----------------------
                    elsif opcode_seg = b"001" then
                        -- use immediate instruction
format
                        -- (op, rs, rt, immediate)

                        rs := bton(current_inst(25 downto
21));
                        rt := bton(current_inst(20 downto
16));
                        immed := current_inst(15 downto
0);

                        case opcode is
                          when i_addi =>
                            inst <= op_addi;
                            if rt /= 0 then
                                add_ovf(reg(rs),
se16to32(immed), reg(rt), ovrflw);
                                if ovrflw = '1' then
                                    exception_flag :=
overflow;
                                end if;
                            end if;

                          when i_addiu =>
                            inst <= op_addiu;
                            if rt /= 0 then
                                reg(rt) := reg(rs) +
se16to32(immed);
                            end if;

                          when i_slti =>
                            inst <= op_slti;
                            if rt /= 0 then
                                if btoi(reg(rs)) <
btoi(se16to32(immed)) then
                                    reg(rt) :=
x"0000_0001";
                                else
                                    reg(rt) :=
x"0000_0000";
                                end if;
                            end if;

                          when i_sltiu =>
                            inst <= op_sltiu;
                            if rt /= 0 then
                                if bton(reg(rs)) <
bton(se16to32(immed)) then
                                    reg(rt) :=
x"0000_0001";
                                else
                                    reg(rt) :=
x"0000_0000";
                                end if;
                            end if;

                          when i_andi =>
                            inst <= op_andi;
                            if rt /= 0 then
                                reg(rt) := (x"0000" &
immed) and reg(rs);
                            end if;

                          when i_ori =>
                            inst <= op_ori;
                            if rt /= 0 then
                                reg(rt) := (x"0000" &
immed) or reg(rs);
                            end if;

                          when i_xori =>
                            inst <= op_xori;
                            if rt /= 0 then
                                reg(rt) := (x"0000" &
immed) xor reg(rs);
                            end if;

                          when i_lui =>
                            inst <= op_lui;
                            if rt /= 0 then
                                ui := immed;
                                reg(rt) := ui & x"0000";
                            end if;

                          when others =>
                            null;

                        end case;


                    ------------------------------------
-----------------------
                    -- load and store instructions
                    ------------------------------------
-----------------------
                    elsif opcode_seg = b"100" or
opcode_seg = b"101" then
                        -- use immediate instruction
format
                        -- (op, base, rt, offset)

                        base := bton(current_inst(25
downto 21));
                        rt := bton(current_inst(20 downto
16));
                        offset := current_inst(15 downto
0);

                        case opcode is
                          when i_lb =>
                            inst <= op_lb;
                            if rt /= 0 then
                                ea := se16to32(offset) +
reg(base);
                                mem_read(ea, byte,
temp_reg_val_1);
                                temp_reg_num_1 := rt;
                                latency_flag := set;
                            end if;

                          when i_lh =>
                            inst <= op_lh;
                            if rt /= 0 then
                                ea := se16to32(offset) +
reg(base);
                                mem_read(ea, halfword,
temp_reg_val_1);
                                temp_reg_num_1 := rt;
                                latency_flag := set;
                            end if;

                          when i_lwl =>
                            inst <= op_lwl;
```

```
                    if rt /= 0 then
                        ea := se16to32(offset) +
reg(base);
                        mem_read(ea, lefty,
temp_reg_val_1);
                        temp_reg_num_1 := rt;
                        temp_addr_1 := ea(1
downto 0);
                        latency_flag := set;
                        left_flag := set;
                    end if;

                when i_lw =>
                    inst <= op_lw;
                    if rt /= 0 then
                        ea := se16to32(offset) +
reg(base);
                        mem_read(ea, word,
temp_reg_val_1);
                        temp_reg_num_1 := rt;
                        latency_flag := set;
                    end if;

                when i_lbu =>
                    inst <= op_lbu;
                    if rt /= 0 then
                        ea := se16to32(offset) +
reg(base);
                        mem_read(ea, ubyte,
temp_reg_val_1);
                        temp_reg_num_1 := rt;
                        latency_flag := set;
                    end if;

                when i_lhu =>
                    inst <= op_lhu;
                    if rt /= 0 then
                        ea := se16to32(offset) +
reg(base);
                        mem_read(ea, uhalfword,
temp_reg_val_1);
                        temp_reg_num_1 := rt;
                        latency_flag := set;
                    end if;

                when i_lwr =>
                    inst <= op_lwr;
                    if rt /= 0 then
                        ea := se16to32(offset) +
reg(base);
                        mem_read(ea, righty,
temp_reg_val_1);
                        temp_reg_num_1 := rt;
                        temp_addr_1 := ea(1
downto 0);
                        latency_flag := set;
                        right_flag := set;
                    end if;

                when i_sb =>
                    inst <= op_sb;
                    ea := se16to32(offset) +
reg(base);
                    mem_write(ea, byte, reg(rt));

                when i_sh =>
                    inst <= op_sh;
                    ea := se16to32(offset) +
reg(base);
                    mem_write(ea, halfword,
reg(rt));

                when i_swl =>
                    inst <= op_swl;
                    ea := se16to32(offset) +
reg(base);
                    mem_write(ea, lefty,
reg(rt));

                when i_sw =>
                    inst <= op_sw;
                    ea := se16to32(offset) +
reg(base);
                    mem_write(ea, word, reg(rt));

                when i_swr =>
                    inst <= op_swr;
                    ea := se16to32(offset) +
reg(base);
                    mem_write(ea, righty,
reg(rt));

                when others =>
                    -- error, reserved
instruction
                    inst <= reserved;
                    exception_flag :=
reserved_inst;

                end case;

                -----------------------------------
----------------------
                -- halt instruction (not a mips
instruction)
                -----------------------------------
----------------------
                elsif opcode = i_halt then
                    inst <= op_halt;
                    run_mode_flag := no;

                -----------------------------------
----------------------
                -- instructions not implemented
                -----------------------------------
----------------------
                elsif (opcode_seg = b"010" or
opcode_seg = b"110" or
                        opcode_seg = b"111") and
opcode(2) = '0' then
                    inst <= not_implmt;
                    exception_flag := not_implmt;


                -----------------------------------
----------------------
                -- reserved instructions
                -----------------------------------
----------------------
                else
                    inst <= reserved;
                    exception_flag := reserved_inst;

                end if;


                -----------------------------------
----------------------
                -- exception handling
                -----------------------------------
----------------------
                if exception_flag /= idle then
                    if bd_flag = yes then
                        cause_reg(31) := '1';
                        epc_reg := pc_reg -
x"0000_0004";
                    else
                        cause_reg(31) := '0';
                        epc_reg := pc_reg;
                    end if;
                    case exception_flag is

                      when reset => null;

                      when breakpt_trap =>
                        cause_reg(5 downto 2) := bp;

                      when syscall_trap =>
                        cause_reg(5 downto 2) := sys;

                      when overflow =>
                        cause_reg(5 downto 2) := ovf;

                      when addr_load =>
                        cause_reg(5 downto 2) :=
adel;

                      when addr_store =>
                        cause_reg(5 downto 2) :=
ades;

                      when bus_inst => null;

                      when bus_data => null;

                      when reserved_inst =>
                        cause_reg(5 downto 2) := ri;

                      when ext_int =>
                        cause_reg(5 downto 2) := int;

                      when not_implmt =>
                        cause_reg(5 downto 2) := ri;

                      when others =>
```

```
                    null;

                end case;
                run_mode_flag := no;


            else
            ------------------------------------
-----------------------
            -- delay slot
            ------------------------------------
-----------------------
            if delay_slot_flag /= ignore then
                if delay_slot_flag = set then
                    delay_slot_flag :=
delay_slot;
                    -- increment pc
                    pc_reg := pc_reg +
x"0000_0004";
                else
                    delay_slot_flag := ignore;
                    pc_reg := pc_temp;

                end if;
            else
                -- increment pc
                pc_reg := pc_reg + x"0000_0004";
            end if;


            ------------------------------------
-----------------------
            -- latency of one instruction on
memory to register loads
            ------------------------------------
-----------------------
            if latency_flag /= ignore then
                if latency_flag = set then
                    temp_reg_val_2 :=
temp_reg_val_1;
                    temp_reg_num_2 :=
temp_reg_num_1;
                    if left_flag = set then
                        temp_addr_2 :=
temp_addr_1;
                        left_flag := delay_slot;
                    end if;
                    if right_flag = set then
                        temp_addr_2 :=
temp_addr_1;
                        right_flag := delay_slot;
                    end if;
                    latency_flag := delay_slot;
                else
                    if left_flag = delay_slot
then
                        case temp_addr_2 is
                          when b"00" =>
                            reg(temp_reg_num_2)
:= temp_reg_val_2;
                          when b"01" =>

reg(temp_reg_num_2)(31 downto 8) :=

temp_reg_val_2(23 downto 0);
                          when b"10" =>

reg(temp_reg_num_2)(31 downto 16) :=

temp_reg_val_2(15 downto 0);
                          when b"11" =>

reg(temp_reg_num_2)(31 downto 24) :=
                                        temp_reg_val_2(7
downto 0);
                        end case;
                        left_flag := ignore;
                    elsif right_flag = delay_slot
then
                        case temp_addr_2 is
                          when b"00" =>
                            reg(temp_reg_num_2)(7
downto 0) :=

temp_reg_val_2(31 downto 24);
                          when b"01" =>

reg(temp_reg_num_2)(15 downto 0) :=

temp_reg_val_2(31 downto 16);
                          when b"10" =>

reg(temp_reg_num_2)(23 downto 0) :=
temp_reg_val_2(31 downto 8);
                          when b"11" =>
                            reg(temp_reg_num_2)
:= temp_reg_val_2;
                        end case;
                        right_flag := ignore;
                    else
                        reg(temp_reg_num_2) :=
temp_reg_val_2;
                    end if;
                latency_flag := ignore;
                end if;
            end if;


            ------------------------------------
-----------------------
            -- set branch delay flag
            ------------------------------------
-----------------------
            if delay_slot_flag = delay_slot then
                bd_flag := yes;
            else
                bd_flag := no;
            end if;

            end if;


            ------------------------------------
-----------------------
            -- update signals
            ------------------------------------
-----------------------
            pc <= pc_reg;          -- program
counter

            r0 <= reg(0);          -- registers r0
thru r31
            r1 <= reg(1);
            r2 <= reg(2);
            r3 <= reg(3);
            r4 <= reg(4);
            r5 <= reg(5);
            r6 <= reg(6);
            r7 <= reg(7);
            r8 <= reg(8);
            r9 <= reg(9);
            r10 <= reg(10);
            r11 <= reg(11);
            r12 <= reg(12);
            r13 <= reg(13);
            r14 <= reg(14);
            r15 <= reg(15);
            r16 <= reg(16);
            r17 <= reg(17);
            r18 <= reg(18);
            r19 <= reg(19);
            r20 <= reg(20);
            r21 <= reg(21);
            r22 <= reg(22);
            r23 <= reg(23);
            r24 <= reg(24);
            r25 <= reg(25);
            r26 <= reg(26);
            r27 <= reg(27);
            r28 <= reg(28);
            r29 <= reg(29);
            r30 <= reg(30);
            r31 <= reg(31);

            hi <= hi_reg;          -- hi, lo
registers for mult and div
            lo <= lo_reg;

            epc <= epc_reg;        -- exception
handling
            cause <= cause_reg;
            exception_sig <= exception_flag;

            -- compare machine state with
expected results
            compare_control_sig <= test;
            wait until compare_ack_sig = yes;
            compare_control_sig <= reset;
            wait until compare_ack_sig = no;

        end loop;     -- end while run_mode_flag =
yes;

        when others =>
```

```
            null;                   -- illegal system
control command
        end case;                       -- end case for
system command

    end process processor;


end cpu_a;
```

COMPONENT:	**MEMORY**
FILENAME:	**MEMORY_E.VHDL**
DESCRIPTION:	**Memory component of behavioral model test bench (entity)**

```
-- library and use clauses
library mgc_portable, ieee;
use mgc_portable.qsim_logic.all;
use mgc_portable.qsim_relations.all;
use ieee.std_logic_1164.all;

use std.textio.all;

library my_packages;
use my_packages.package_1.all;


entity memory is
    port(mem_control_sig:   in    mem_control_type;
         addr_bus:          in    bit_30;
         addr_bus_lo:       in    bit_2;
         mem_ack_sig:       out   question_type;
         mem_exception_sig: out   question_type;
         data_bus:          inout bus_bit_32 bus);
end memory;
```

COMPONENT:	**MEMORY**
FILENAME:	**MEMORY_A.VHDL**
DESCRIPTION:	**Memory component of behavioral model test bench (architecture)**

```
architecture memory_a of memory is
begin

    memory: process
        constant low_address: integer := 0;
        constant high_address: integer := 65535;
        type memory_array is
            array (integer range low_address to
high_address) of bit_32;
        variable mem: memory_array;
        variable addr: integer range 0 to high_address;
        variable temp: bit_32;

        file in1: text is in "machine";
        variable line1: line;
        variable inst: bit_32;


    begin

      wait on mem_control_sig;      -- entry into memory
process

      data_bus <= null;

      if mem_control_sig /= reset or mem_control_sig /=
load then
          assert addr_bus(29 downto 14) = x"0000"
              report "ADDRESS OUT OF RANGE";
      end if;

      if addr_bus(29 downto 14) = x"0000" then
          case mem_control_sig is
            when read_w =>
              addr := bton(addr_bus);
              case addr_bus_lo is
                when b"00" =>
                  data_bus <= mem(addr);

                when others =>
                  -- flag address exception
                  mem_exception_sig <= yes;
              end case;
              mem_ack_sig <= yes after delay;

            when read_hw =>
              addr := bton(addr_bus);
              temp := mem(addr);
              case addr_bus_lo is
                when b"00" =>
                  data_bus <= se16to32(temp(31 downto
16));
                when b"10" =>
                  data_bus <= se16to32(temp(15 downto
0));
                when others =>
                  -- flag address exception
                  mem_exception_sig <= yes;
              end case;
              mem_ack_sig <= yes after delay;

            when read_uhw =>
              addr := bton(addr_bus);
              temp := mem(addr);
              case addr_bus_lo is
                when b"00" =>
                  data_bus <= x"0000" & temp(31 downto
16);
                when b"10" =>
                  data_bus <= x"0000" & temp(15 downto
0);
                when others =>
                  -- flag address exception
                  mem_exception_sig <= yes;
              end case;
              mem_ack_sig <= yes after delay;

            when read_b =>
              addr := bton(addr_bus);
              temp := mem(addr);
              case addr_bus_lo is
                when b"00" =>
                  data_bus <= se8to32(temp(31 downto
24));
                when b"01" =>
                  data_bus <= se8to32(temp(23 downto
16));
                when b"10" =>
                  data_bus <= se8to32(temp(15 downto
8));
                when b"11" =>
                  data_bus <= se8to32(temp(7 downto
0));
              end case;
              mem_ack_sig <= yes after delay;

            when read_ub =>
              addr := bton(addr_bus);
              temp := mem(addr);
              case addr_bus_lo is
                when b"00" =>
                  data_bus <= x"000000" & temp(31
downto 24);
                when b"01" =>
                  data_bus <= x"000000" & temp(23
downto 16);
                when b"10" =>
                  data_bus <= x"000000" & temp(15
downto 8);
                when b"11" =>
                  data_bus <= x"000000" & temp(7 downto
0);
              end case;
              mem_ack_sig <= yes after delay;

            when read_l =>
              addr := bton(addr_bus);
              temp := mem(addr);
              case addr_bus_lo is
                when b"00" =>
                  data_bus <= temp;
                when b"01" =>
                  data_bus(23 downto 0) <= temp(23
downto 0);
                  data_bus(31 downto 24) <= x"00";
                when b"10" =>
                  data_bus(15 downto 0) <= temp(15
downto 0);
                  data_bus(31 downto 16) <= x"0000";
                when b"11" =>
                  data_bus(7 downto 0) <= temp(7 downto
0);
                  data_bus(31 downto 8) <= x"000000";
              end case;
              mem_ack_sig <= yes after delay;

            when read_r =>
```

```
            addr := bton(addr_bus);
            temp := mem(addr);
            case addr_bus_lo is
              when b"00" =>
                data_bus(31 downto 24) <= temp(31
downto 24);
                data_bus(23 downto 0) <= x"000000";
              when b"01" =>
                data_bus(31 downto 16) <= temp(31
downto 16);
                data_bus(15 downto 0) <= x"0000";
              when b"10" =>
                data_bus(31 downto 8) <= temp(31
downto 8);
                data_bus(7 downto 0) <= x"00";
              when b"11" =>
                data_bus <= temp;
            end case;
            mem_ack_sig <= yes after delay;

          when write_w =>
            addr := bton(addr_bus);
            case addr_bus_lo is
              when b"00" =>
                --mem(addr) := data_bus;
                  mem(addr) := data_bus;
              when others =>
                -- flag address exception
                mem_exception_sig <= yes;
            end case;
            mem_ack_sig <= yes after delay;

          when write_hw =>
            addr := bton(addr_bus);
            case addr_bus_lo is
              when b"00" =>
                mem(addr)(31 downto 16) :=
data_bus(15 downto 0);
              when b"10" =>
                mem(addr)(15 downto 0) := data_bus(15
downto 0);
              when others =>
                -- flag address exception
                mem_exception_sig <= yes;
            end case;
            mem_ack_sig <= yes after delay;

          when write_b =>
            addr := bton(addr_bus);
            case addr_bus_lo is
              when b"00" =>
                mem(addr)(31 downto 24) := data_bus(7
downto 0);
              when b"01" =>
                mem(addr)(23 downto 16) := data_bus(7
downto 0);
              when b"10" =>
                mem(addr)(15 downto 8) := data_bus(7
downto 0);
              when b"11" =>
                mem(addr)(7 downto 0) := data_bus(7
downto 0);
            end case;
            mem_ack_sig <= yes after delay;

          when write_l =>
            addr := bton(addr_bus);
            case addr_bus_lo is
              when b"00" =>
                mem(addr) := data_bus;
              when b"01" =>
                mem(addr)(23 downto 0) := data_bus(31
downto 8);
              when b"10" =>
                mem(addr)(15 downto 0) := data_bus(31
downto 16);
              when b"11" =>
                mem(addr)(7 downto 0) := data_bus(31
downto 24);
            end case;
            mem_ack_sig <= yes after delay;

          when write_r =>
            addr := bton(addr_bus);
            case addr_bus_lo is
              when b"00" =>
                mem(addr)(31 downto 24) := data_bus(7
downto 0);
              when b"01" =>
                mem(addr)(31 downto 16) :=
data_bus(15 downto 0);
              when b"10" =>
                mem(addr)(31 downto 8) := data_bus(23
downto 0);
              when b"11" =>
                mem(addr) := data_bus;
            end case;
            mem_ack_sig <= yes after delay;

          when load =>
            addr := 0;
            while endfile(in1) = false loop
                readline(in1, line1);
                read(line1, inst);
                mem(addr) := inst;
                addr := addr + 1;
            end loop;
            mem_ack_sig <= yes after delay;

          when reset =>
            mem_ack_sig <= no after delay;

          when others =>
            null;                    -- error, illegal
operation
          end case;
        end if;

    end process memory;

end memory_a;
```

COMPONENT:	**COMPARE**
FILENAME:	**COMPARE_E.VHDL**
DESCRIPTION:	**Compare component of behavioral model test bench (entity)**

```
-- library and use clauses
library mgc_portable, ieee;
use mgc_portable.qsim_logic.all;
use mgc_portable.qsim_relations.all;
use ieee.std_logic_1164.all;

use std.textio.all;

library my_packages;
use my_packages.package_1.all;

entity compare is
    port(compare_control_sig: in
compare_control_type;
         pc:                  in    bit_32;
         r1:                  in    bit_32;
         r2:                  in    bit_32;
         r3:                  in    bit_32;
         r4:                  in    bit_32;
         r31:                 in    bit_32;
         hi:                  in    bit_32;
         lo:                  in    bit_32;
         epc:                 in    bit_32;
         cause:               in    bit_32;
         compare_ack_sig:     out   question_type);
end compare;
```

COMPONENT:	**COMPARE**
FILENAME:	**COMPARE_A.VHDL**
DESCRIPTION:	**Compare component of behavioral model test bench (architecture)**

```
architecture compare_a of compare is
begin

    compare: process
        type test_field_type is array (0 to 9) of
bit_32;
        type test_array_type is array (0 to 1000) of
test_field_type;
        variable expect: test_array_type;
        variable index: integer range 0 to 1000;

        file in1: text is in "expected";
        variable line1: line;
        variable inst: bit_32;
    begin
        wait on compare_control_sig;
        case compare_control_sig is
          when load =>
```

```
            index := 0;
            while endfile(in1) = false loop
                for i in 0 to 9 loop
                    if endfile(in1) = false then
                        readline(in1, line1);
                        read(line1, inst);
                        expect(index)(i) := inst;
                    end if;
                end loop;
                index := index + 1;
            end loop;
            index := 0;
            compare_ack_sig <= yes after delay;

          when test =>
            assert pc = expect(index)(0)
                report "PC REGISTER ERROR"
                severity warning;
            assert r1 = expect(index)(1)
                report "REGISTER 1 ERROR"
                severity warning;
            assert r2 = expect(index)(2)
                report "REGISTER 2 ERROR"
                severity warning;
            assert r3 = expect(index)(3)
                report "REGISTER 3 ERROR"
                severity warning;
            assert r4 = expect(index)(4)
                report "REGISTER 4 ERROR"
                severity warning;
            assert r31 = expect(index)(5)
                report "REGISTER 31 ERROR"
                severity warning;
            assert hi = expect(index)(6)
                report "REGISTER HI ERROR"
                severity warning;
            assert lo = expect(index)(7)
                report "REGISTER LO ERROR"
                severity warning;
            assert epc = expect(index)(8)
                report "EPC REGISTER ERROR"
                severity warning;
            assert cause = expect(index)(9)
                report "CAUSE REGISTER ERROR"
                severity warning;
            index := index + 1;
            compare_ack_sig <= yes after delay;

          when reset =>
            compare_ack_sig <= no after delay;

          --when stop => null;

          --when error => null;
        end case;

    end process compare;

end compare_a;
```

COMPONENT:	PACKAGE_1
FILENAME:	PACKAGE_1_H.VHDL
DESCRIPTION:	Behavioral model package (header)

```
package package_1 is
    constant delay: time := 10 ns;

    subtype bit_2 is bit_vector (1 downto 0);
    subtype bit_3 is bit_vector (2 downto 0);
    subtype bit_4 is bit_vector (3 downto 0);
    subtype bit_5 is bit_vector (4 downto 0);
    subtype bit_6 is bit_vector (5 downto 0);
    subtype bit_8 is bit_vector(7 downto 0);
    subtype bit_16 is bit_vector (15 downto 0);
    subtype bit_20 is bit_vector (19 downto 0);
    subtype bit_24 is bit_vector(23 downto 0);
    subtype bit_26 is bit_vector (25 downto 0);
    subtype bit_30 is bit_vector (29 downto 0);
    subtype bit_32 is bit_vector (31 downto 0);
    subtype bit_64 is bit_vector(63 downto 0);

    subtype bit_5_range is integer range 0 to 31;
    subtype bit_16_range is integer range -32768 to
32767;
    subtype bit_26_range is integer range -67108864 to
67108863;

    type bit_32_array is array (integer range <>) of
bit_32;

    function wired_or(driver: bit_32_array) return
bit_32;
    subtype bus_bit_32 is wired_or bit_32;

    type sys_control_type is (stop, run, load, reset);
    type mem_control_type is (reset, read_b, read_ub,
read_hw, read_uhw,
                              read_w, read_l, read_r,
write_b, write_hw,
                              write_w, write_l,
write_r, load);
    type question_type is (no, yes);
    type delay_slot_type is (ignore, delay_slot, set);
    type inst_type is (nop, not_implmt, reserved,
op_halt, op_add, op_addi,
                       op_addiu, op_addu, op_and,
op_andi, op_beq, op_bgez,
                       op_bgezal, op_bgtz, op_blez,
op_bltz, op_bltzal, op_bne,
                       op_break, op_div, op_divu, op_j,
op_jal, op_jalr, op_jr,
                       op_lb, op_lbu, op_lh, op_lhu,
op_lui, op_lw, op_lwl,
                       op_lwr, op_mfhi, op_mflo,
op_mthi, op_mtlo, op_mult,
                       op_multu, op_nor, op_or, op_ori,
op_sb, op_sh, op_sll,
                       op_sllv, op_slt, op_slti,
op_sltiu, op_sltu, op_sra,
                       op_srav, op_srl, op_srlv,
op_sub, op_subu, op_sw, op_swl,
                       op_swr, op_syscall, op_xor,
op_xori);
    type size_type is (byte, ubyte, halfword,
uhalfword, word, lefty, righty);
    type compare_control_type is (reset, load, test);


    ------------------------------------------------
------ ----------------
    -- constants for implemented instructions
    ------------------------------------------------
----------------------
    constant i_special: bit_6 := o"00";
    constant i_bcond:   bit_6 := o"01";
    constant i_j:       bit_6 := o"02";
    constant i_jal:     bit_6 := o"03";
    constant i_beq:     bit_6 := o"04";
    constant i_bne:     bit_6 := o"05";
    constant i_blez:    bit_6 := o"06";
    constant i_bgtz:    bit_6 := o"07";

    constant i_addi:    bit_6 := o"10";
    constant i_addiu:   bit_6 := o"11";
    constant i_slti:    bit_6 := o"12";
    constant i_sltiu:   bit_6 := o"13";
    constant i_andi:    bit_6 := o"14";
    constant i_ori:     bit_6 := o"15";
    constant i_xori:    bit_6 := o"16";
    constant i_lui:     bit_6 := o"17";

    constant i_lb:      bit_6 := o"40";
    constant i_lh:      bit_6 := o"41";
    constant i_lwl:     bit_6 := o"42";
    constant i_lw:      bit_6 := o"43";
    constant i_lbu:     bit_6 := o"44";
    constant i_lhu:     bit_6 := o"45";
    constant i_lwr:     bit_6 := o"46";

    constant i_sb:      bit_6 := o"50";
    constant i_sh:      bit_6 := o"51";
    constant i_swl:     bit_6 := o"52";
    constant i_sw:      bit_6 := o"53";
    constant i_swr:     bit_6 := o"56";

    constant i_sll:     bit_6 := o"00";
    constant i_srl:     bit_6 := o"02";
    constant i_sra:     bit_6 := o"03";
    constant i_sllv:    bit_6 := o"04";
    constant i_srlv:    bit_6 := o"06";
    constant i_srav:    bit_6 := o"07";

    constant i_jr:      bit_6 := o"10";
    constant i_jalr:    bit_6 := o"11";
    constant i_syscall: bit_6 := o"14";
    constant i_break:   bit_6 := o"15";

    constant i_mfhi:    bit_6 := o"20";
    constant i_mthi:    bit_6 := o"21";
    constant i_mflo:    bit_6 := o"22";
    constant i_mtlo:    bit_6 := o"23";

    constant i_mult:    bit_6 := o"30";
```

```
    constant i_multu:   bit_6 := o"31";
    constant i_div:     bit_6 := o"32";
    constant i_divu:    bit_6 := o"33";

    constant i_add:     bit_6 := o"40";
    constant i_addu:    bit_6 := o"41";
    constant i_sub:     bit_6 := o"42";
    constant i_subu:    bit_6 := o"43";
    constant i_and:     bit_6 := o"44";
    constant i_or:      bit_6 := o"45";
    constant i_xor:     bit_6 := o"46";
    constant i_nor:     bit_6 := o"47";

    constant i_slt:     bit_6 := o"52";
    constant i_sltu:    bit_6 := o"53";

    constant i_bltz:    bit_5 := b"00_000";
    constant i_bgez:    bit_5 := b"00_001";

    constant i_bltzal:  bit_5 := b"10_000";
    constant i_bgezal:  bit_5 := b"10_001";

    constant i_halt:      bit_6 := o"77";


    ------------------------------------------------
----------------------
    -- exception stuff
    ------------------------------------------------
----------------------
    type exception_type is (idle, reset, breakpt_trap,
syscall_trap, overflow,
                            addr_load, addr_store,
bus_inst, bus_data,
                            reserved_inst, ext_int,
not_implmt);

    constant int:  bit_4 := x"0";
    constant adel: bit_4 := x"4";
    constant ades: bit_4 := x"5";
    constant ibe:  bit_4 := x"6";
    constant dbe:  bit_4 := x"7";
    constant sys:  bit_4 := x"8";
    constant bp:   bit_4 := x"9";
    constant ri:   bit_4 := x"a";
    constant ovf:  bit_4 := x"c";


    ------------------------------------------------
------------------ ---
    -- declaration of functions and procedures
    ------------------------------------------------
------------------ --
    function btoi(bits: in bit_vector) return integer;
    function bton(bits: in bit_vector) return integer;
    procedure itob(int: in integer; bits: out
bit_vector);
    function shift_ll(arg_in: in bit_vector; amt: in
integer) return bit_vector;
    function shift_rl(arg_in: in bit_vector; amt: in
integer) return bit_vector;
    function shift_ra(arg_in: in bit_vector; amt: in
integer) return bit_vector;
    function se16to32(arg_in: in bit_16) return bit_32;
    function se8to32(arg_in: in bit_8) return bit_32;
    function se24to32(arg_in: in bit_24) return bit_32;
    function se16to30(arg_in: in bit_16) return bit_30;
    function "+"(l, r: in bit_32) return bit_32;
    function add_64(l, r: in bit_64) return bit_64;
    procedure add_ovf(l, r: in bit_32; s: out bit_32;
c: out bit);
    function "-"(l, r: in bit_32) return bit_32;
    procedure sub_ovf(l, r: in bit_32; s: out bit_32;
c: out bit);
    function "*"(t, b: in bit_32) return bit_64;
    function mult(a, b: in bit_32) return bit_64;
    function "/"(dividend, divs: in bit_32) return
bit_64;
    function div(a, b: in bit_32) return bit_64;


end package_1;
```

COMPONENT:	PACKAGE_1
FILENAME:	PACKAGE_1_B.VHDL
DESCRIPTION:	Behavioral model package (body)

```
package body package_1 is

    function wired_or(driver: bit_32_array) return
bit_32 is
        variable result: bit_32 := x"0000_0000";
    begin
        for i in driver'range loop
            result := result or driver(i);
        end loop;
        return result;
    end wired_or;


    function btoi(bits: in bit_vector) return integer
is
        variable temp: bit_vector(bits'range);
        variable result: integer := 0;
    begin
        if bits(bits'left) = '1' then      -- negative
number
            temp := not bits;
        else
            temp := bits;
        end if;
        for index in bits'range loop
            result := result * 2 +
bit'pos(temp(index));
        end loop;
        if bits(bits'left) = '1' then
            result := (-result) - 1;
        end if;
        return result;
    end btoi;


    function bton(bits: in bit_vector) return integer
is
        variable result: natural := 0;
    begin
        for index in bits'range loop
            result := result * 2 +
bit'pos(bits(index));
        end loop;
        return result;
    end bton;


    procedure itob(int: in integer; bits: out
bit_vector) is
        variable temp: integer;
        variable result: bit_vector(bits'range);
    begin
        if int < 0 then
            temp := -(int + 1);
        else
            temp := int;
        end if;
        for index in bits'reverse_range loop
            result(index) := bit'val(temp rem 2);
            temp := temp / 2;
        end loop;
        if int < 0 then
            result := not result;
            result(bits'left) := '1';
        end if;
        bits := result;
    end itob;


    function shift_ll(arg_in: in bit_vector; amt: in
integer)
                      return bit_vector is
        variable result: bit_vector(arg_in'range);
    begin
        result := arg_in;
        for j in 1 to amt loop
            for index in arg_in'range loop
                if index /= 0 then
                    result(index) := result(index - 1);
                else
                    result(index) := '0';
                end if;
            end loop;
        end loop;
        return result;
    end shift_ll;


    function shift_rl(arg_in: in bit_vector; amt: in
integer)
                      return bit_vector is
        variable result: bit_vector(arg_in'range);
    begin
        result := arg_in;
        for j in 1 to amt loop
```

```
            for index in arg_in'reverse_range loop
                if index /= arg_in'high then
                    result(index) := result(index + 1);
                else
                    result(index) := '0';
                end if;
            end loop;
        end loop;
        return result;
    end shift_rl;


    function shift_ra(arg_in: in bit_vector; amt: in
integer)
                    return bit_vector is
        variable result: bit_vector(arg_in'range);
    begin
        result := arg_in;
        if arg_in(arg_in'left) = '1' then
            for j in 1 to amt loop
                for index in arg_in'reverse_range loop
                    if index /= arg_in'high then
                        result(index) := result(index +
1);
                    else
                        result(index) := '1';
                    end if;
                end loop;
            end loop;
        else
            for j in 1 to amt loop
                for index in arg_in'reverse_range loop
                    if index /= arg_in'high then
                        result(index) := result(index +
1);
                    else
                        result(index) := '0';
                    end if;
                end loop;
            end loop;
        end if;
        return result;
    end shift_ra;



    function se16to32(arg_in: in bit_16) return bit_32
is
        variable result: bit_32;
    begin
        if arg_in(arg_in'left) = '1' then
            result := x"ffff" & arg_in;
        else
            result := x"0000" & arg_in;
        end if;
        return result;
    end se16to32;


    function se8to32(arg_in: in bit_8) return bit_32 is
        variable result: bit_32;
    begin
        if arg_in(arg_in'left) = '1' then
            result := b"1111_1111_1111_1111_1111_1111"
& arg_in;
        else
            result := b"0000_0000_0000_0000_0000_0000"
& arg_in;
        end if;
        return result;
    end se8to32;


    function se24to32(arg_in: in bit_24) return bit_32
is
        variable result: bit_32;
    begin
        if arg_in(arg_in'left) = '1' then
            result := b"1111_1111" & arg_in;
        else
            result := b"0000_0000" & arg_in;
        end if;
        return result;
    end se24to32;


    function se16to30(arg_in: in bit_16) return bit_30
is
        variable result: bit_30;
    begin
        if arg_in(arg_in'left) = '1' then
            result := b"11_1111_1111_1111" & arg_in;
        else
            result := b"00_0000_0000_0000" & arg_in;
        end if;
        return result;
    end se16to30;


    function "+"(l, r: in bit_32) return bit_32 is
        variable carry: bit := '0';
        variable sum: bit_32;
    begin
        for i in 0 to 31 loop
            if carry = '0' then
                carry := l(i) and r(i);
                sum(i) := l(i) xor r(i);
            else
                carry := l(i) or r(i);
                sum(i) := not( l(i) xor r(i) );
            end if;
        end loop;
        return sum;
    end "+";


    function add_64(l, r: in bit_64) return bit_64 is
        variable carry: bit := '0';
        variable sum: bit_64;
    begin
        for i in 0 to 63 loop
            if carry = '0' then
                carry := l(i) and r(i);
                sum(i) := l(i) xor r(i);
            else
                carry := l(i) or r(i);
                sum(i) := not( l(i) xor r(i) );
            end if;
        end loop;
        return sum;
    end add_64;


    procedure add_ovf(l, r: in bit_32; s: out bit_32;
c: out bit) is
        variable sum: bit_32;
        variable carry: bit;
        variable carry30: bit;
        variable carry31: bit;
    begin
        for i in 0 to 31 loop
            if carry = '0' then
                carry := l(i) and r(i);
                sum(i) := l(i) xor r(i);
            else
                carry := l(i) or r(i);
                sum(i) := not( l(i) xor r(i) );
            end if;
            if i = 30 then
                carry30 := carry;
            elsif i = 31 then
                carry31 := carry;
            end if;
        end loop;
        c := carry30 xor carry31;
        s := sum;
    end add_ovf;


    function "-"(l, r: in bit_32) return bit_32 is
        variable temp, result: bit_32;
    begin
        temp := (not r) + x"0000_0001";
        result := l + temp;
        return result;
    end "-";


    procedure sub_ovf(l, r: in bit_32; s: out bit_32;
c: out bit) is
        variable temp, result: bit_32;
    begin
        temp := (not r) + x"0000_0001";
        add_ovf(l, temp, s, c);
    end sub_ovf;


    function "*"(t, b: in bit_32) return bit_64 is
        type mult_array is array (0 to 31) of bit_64;
        variable ma: mult_array;
        variable carry: bit := '0';
    begin
        for bot in 0 to 31 loop
            if b(bot) = '0' then
                ma(bot) := x"0000_0000_0000_0000";
            else
```

```
                ma(bot)(31 downto 0) := t;
                ma(bot) := shift_ll(ma(bot), bot);
            end if;
        end loop;
        for j in 0 to 30 loop
            carry := '0';
            for i in 0 to 63 loop
                if carry = '0' then
                    carry := ma(j)(i) and ma(j + 1)(i);
                    ma(j + 1)(i) := ma(j)(i) xor ma(j +
1)(i);
                else
                    carry := ma(j)(i) or ma(j + 1)(i);
                    ma(j + 1)(i) := not( ma(j)(i) xor
ma(j + 1)(i) );
                end if;
            end loop;
        end loop;

        return ma(31);
    end "*";


    function mult(a, b: in bit_32) return bit_64 is
        variable temp: bit_64;
        variable a_twos_comp: bit_32;
        variable b_twos_comp: bit_32;
    begin
        if a(31) = '0' and b(31) = '0' then
            temp := a * b;
        elsif a(31) = '0' and b(31) = '1' then
            b_twos_comp := (not b) + x"0000_0001";
            temp := a * b_twos_comp;
            temp := add_64(not temp,
x"0000_0000_0000_0001");
        elsif a(31) = '1' and b(31) = '0' then
            a_twos_comp := (not a) + x"0000_0001";
            temp := a_twos_comp * b;
            temp := add_64(not temp,
x"0000_0000_0000_0001");
        else
            a_twos_comp := (not a) + x"0000_0001";
            b_twos_comp := (not b) + x"0000_0001";
            temp := a_twos_comp * b_twos_comp;
        end if;
        return temp;
    end mult;


    function "/"(dividend, divs: in bit_32) return
bit_64 is
        variable divd_temp: bit_32;
        variable divs_temp: bit_32;
        variable quot: bit_32;
        variable result: bit_64;
        variable divd: bit_32;
    begin
        divd := dividend;
        for i in 31 downto 0 loop
            divd_temp := shift_rl(divd, i);
            if divs > divd_temp then
                quot(i) := '0';
            else
                quot(i) := '1';
                divs_temp := shift_ll(divs, i);
                divd := divd - divs_temp;
            end if;
        end loop;
        result(31 downto 0) := quot;
        result(63 downto 32) := divd;
        return result;
    end "/";


    function div(a, b: in bit_32) return bit_64 is
        variable temp: bit_64;
        variable a_twos_comp: bit_32;
        variable b_twos_comp: bit_32;
    begin
        if a(31) = '0' and b(31) = '0' then
            temp := a / b;
        elsif a(31) = '0' and b(31) = '1' then
            b_twos_comp := (not b) + x"0000_0001";
            temp := a / b_twos_comp;
            temp(31 downto 0) := (not temp(31 downto
0)) + x"0000_0001";
        elsif a(31) = '1' and b(31) = '0' then
            a_twos_comp := (not a) + x"0000_0001";
            temp := a_twos_comp / b;
            temp(31 downto 0) := (not temp(31 downto
0)) + x"0000_0001";
        else
            a_twos_comp := (not a) + x"0000_0001";
            b_twos_comp := (not b) + x"0000_0001";
            temp := a_twos_comp / b_twos_comp;
        end if;
        return temp;
    end div;

end package_1;
```

APPENDIX C - DATAFLOW MODEL SOURCE CODE

COMPONENT:	DFMIPS
FILENAME:	DFMIPS_E.VHDL
DESCRIPTION:	Test bench for dataflow model of asynchronous version of MIPS R3000 microprocessor (entity)

```
library my_packages;
use my_packages.package_1.all;
use my_packages.dfpack.all;

library df_comp;
use df_comp.dfcpu.all;
use df_comp.dfmemory.all;
use df_comp.dfcompare.all;

entity dfmips is
end dfmips;
```

COMPONENT:	DFMIPS
FILENAME:	DFMIPS_A.VHDL
DESCRIPTION:	Test bench for dataflow model of asynchronous version of MIPS R3000 microprocessor (architecture)

```
architecture dfmips_a of dfmips is
    component dfcpu
        port(sys_control_sig:  in    sys_control_type;
             memory_ack:       in    bit;
             memory_load_ack:  in    question_type;
             compare_ack:      in    question_type;
             compare_load_ack: in    question_type;
             addr_bus:         inout bus_bit_32 bus;
             data_bus:         inout bus_bit_32 bus;
             memory_req:       out   bit;
             memory_w:         out   bit;
             memory_opcode:    out   bit_3;
             memory_load:      out   question_type;
             compare:          out   question_type;
             compare_load:     out   question_type;
             pc_test:          out   bit_32;
             r1_test:          out   bit_32;
             r2_test:          out   bit_32;
             r3_test:          out   bit_32;
             r4_test:          out   bit_32;
             r31_test:         out   bit_32;
             hi_test:          out   bit_32;
             lo_test:          out   bit_32);
    end component;

    component dfmemory
        port(load:     in    question_type;
             req:      in    bit;
             w:        in    bit;
             opcode:   in    bit_3;
             ack:      out   bit;
             load_ack: out   question_type;
             addr_bus: inout bus_bit_32 bus;
             data_bus: inout bus_bit_32 bus);
    end component;

    component dfcompare
        port(compare:          in  question_type;
             compare_load:     in  question_type;
             pc_test:          in  bit_32;
             r1_test:          in  bit_32;
             r2_test:          in  bit_32;
             r3_test:          in  bit_32;
             r4_test:          in  bit_32;
             r31_test:         in  bit_32;
             hi_test:          in  bit_32;
             lo_test:          in  bit_32;
             compare_ack:      out question_type;
             compare_load_ack: out question_type);
    end component;

    signal sys_control: sys_control_type;
    signal memory_load, memory_load_ack: question_type;
    signal memory_req, memory_ack, memory_w: bit;
    signal memory_opcode: bit_3;
    signal addr_bus, data_bus: bus_bit_32;

    signal compare_ack, compare_load_ack:
question_type;
    signal compare, compare_load: question_type;
    signal pc_test, r1_test, r2_test, r3_test: bit_32;
    signal r4_test, r31_test, hi_test, lo_test: bit_32;

begin

    cpu_module: dfcpu
        port map(sys_control, memory_ack,
memory_load_ack, compare_ack,
                 compare_load_ack, addr_bus, data_bus,
memory_req,
                 memory_w, memory_opcode, memory_load,
compare,
                 compare_load, pc_test, r1_test,
r2_test, r3_test,
                 r4_test, r31_test, hi_test, lo_test);

    memory_module: dfmemory
        port map(memory_load, memory_req, memory_w,
memory_opcode,
                 memory_ack, memory_load_ack, addr_bus,
data_bus);

    compare_module: dfcompare
        port map(compare, compare_load, pc_test,
r1_test, r2_test,
                 r3_test, r4_test, r31_test, hi_test,
lo_test,
                 compare_ack, compare_load_ack);

end dfmips_a;
```

COMPONENT:	DFPACK
FILENAME:	DFPACK_H.VHDL
DESCRIPTION:	Dataflow model package (header)

```
library my_packages;
use my_packages.package_1.all;

package dfpack is
    subtype bit_10 is bit_vector (9 downto 0);

    constant m_lb:      bit_3 := o"0";
    constant m_lh:      bit_3 := o"1";
    constant m_lwl:     bit_3 := o"2";
    constant m_lw:      bit_3 := o"3";
    constant m_lbu:     bit_3 := o"4";
    constant m_lhu:     bit_3 := o"5";
    constant m_lwr:     bit_3 := o"6";

    constant m_sb:      bit_3 := o"0";
    constant m_sh:      bit_3 := o"1";
    constant m_swl:     bit_3 := o"2";
    constant m_sw:      bit_3 := o"3";
    constant m_swr:     bit_3 := o"6";
end dfpack;
```

COMPONENT:	DATAFLOW COMPONENTS
FILENAME:	???_E.VHDL for entity and ???_A.VHDL for architecture
DESCRIPTION:	All components used in dataflow model follow (entity shown first)

DF2TO1MUX

```
entity df2to1mux is
    port(i0: in  bit;
         i1: in  bit;
         s:  in  bit;
```

```
        o:  out bit);
end df2to1mux;


architecture df2to1mux_a of df2to1mux is
begin
    o <= i0 after 0.3 ns when s = '0' else
         i1 after 0.3 ns;
end df2to1mux_a;
```

DF2TO1MUX16

```
library my_packages;
use my_packages.package_1.all;

entity df2to1mux16 is
    port (i0: in  bit_16;
          i1: in  bit_16;
          s:  in  bit;
          o:  out bit_16);
end df2to1mux16;


architecture df2to1mux16_a of df2to1mux16 is
begin
    o <= i0 after 0.3 ns when s = '0' else
         i1 after 0.3 ns;
end df2to1mux16_a;
```

DF2TO1MUX3

```
library my_packages;
use my_packages.package_1.all;

entity df2to1mux3 is
    port (i0: in  bit_3;
          i1: in  bit_3;
          s:  in  bit;
          o:  out bit_3);
end df2to1mux3;


architecture df2to1mux3_a of df2to1mux3 is
begin
    o <= i0 after 0.3 ns when s = '0' else
         i1 after 0.3 ns;
end df2to1mux3_a;
```

DF2TO1MUX30

```
library my_packages;
use my_packages.package_1.all;

entity df2to1mux30 is
    port(i0: in  bit_30;
         i1: in  bit_30;
         s:  in  bit;
         o:  out bit_30);
end df2to1mux30;


architecture df2to1mux30_a of df2to1mux30 is
begin
    o <= i0 after 0.3 ns when s = '0' else
         i1 after 0.3 ns;
end df2to1mux30_a;
```

DF2TO1MUX32

```
library my_packages;
use my_packages.package_1.all;

entity df2to1mux32 is
    port(i0: in  bit_32;
         i1: in  bit_32;
         s:  in  bit;
         o:  out bit_32);
end df2to1mux32;


architecture df2to1mux32_a of df2to1mux32 is
begin
    o <= i0 after 0.3 ns when s = '0' else
         i1 after 0.3 ns;
end df2to1mux32_a;
```

DF2TO1MUX8

```
library my_packages;
use my_packages.package_1.all;

entity df2to1mux8 is
    port (i0: in  bit_8;
          i1: in  bit_8;
          s:  in  bit;
          o:  out bit_8);
end df2to1mux8;


architecture df2to1mux8_a of df2to1mux8 is
begin
    o <= i0 after 0.3 ns when s = '0' else
         i1 after 0.3 ns;
end df2to1mux8_a;
```

DF32TO1MUX

```
library my_packages;
use my_packages.package_1.all;

entity df32to1mux is
    port(i: in bit_32;
         s: in bit_5;
         o: out bit);
end df32to1mux;


architecture df32to1mux_a of df32to1mux is
    signal t: bit;
begin
    t <= i(0) after 3.6 ns when s = b"00000" else
         i(1) after 3.6 ns when s = b"00001" else
         i(2) after 3.6 ns when s = b"00010" else
         i(3) after 3.6 ns when s = b"00011" else
         i(4) after 3.6 ns when s = b"00100" else
         i(5) after 3.6 ns when s = b"00101" else
         i(6) after 3.6 ns when s = b"00110" else
         i(7) after 3.6 ns when s = b"00111" else

         i(8)  after 3.6 ns when s = b"01000" else
         i(9)  after 3.6 ns when s = b"01001" else
         i(10) after 3.6 ns when s = b"01010" else
         i(11) after 3.6 ns when s = b"01011" else
         i(12) after 3.6 ns when s = b"01100" else
         i(13) after 3.6 ns when s = b"01101" else
         i(14) after 3.6 ns when s = b"01110" else
         i(15) after 3.6 ns when s = b"01111" else

         i(16) after 3.6 ns when s = b"10000" else
         i(17) after 3.6 ns when s = b"10001" else
         i(18) after 3.6 ns when s = b"10010" else
         i(19) after 3.6 ns when s = b"10011" else
         i(20) after 3.6 ns when s = b"10100" else
         i(21) after 3.6 ns when s = b"10101" else
         i(22) after 3.6 ns when s = b"10110" else
         i(23) after 3.6 ns when s = b"10111" else

         i(24) after 3.6 ns when s = b"11000" else
         i(25) after 3.6 ns when s = b"11001" else
         i(26) after 3.6 ns when s = b"11010" else
         i(27) after 3.6 ns when s = b"11011" else
         i(28) after 3.6 ns when s = b"11100" else
         i(29) after 3.6 ns when s = b"11101" else
         i(30) after 3.6 ns when s = b"11110" else
         i(31) after 3.6 ns when s = b"11111" else
         t;
    o <= t;
end df32to1mux_a;
```

DF4TO1MUX

```
entity df4to1mux is
    port (i0: in  bit;
          i1: in  bit;
          i2: in  bit;
          i3: in  bit;
          s0: in  bit;
          s1: in  bit;
          o:  out bit);
end df4to1mux;
```

```
architecture df4to1mux_a of df4to1mux is
begin
    o <= i0 after 0.4 ns when s1 = '0' and s0 = '0'
else
         i1 after 0.4 ns when s1 = '0' and s0 = '1'
else
         i2 after 0.4 ns when s1 = '1' and s0 = '0'
else
         i3 after 0.4 ns;
end df4to1mux_a;
```

DF4TO1MUX10

```
library my_packages;
use my_packages.package_1.all;
use my_packages.dfpack.all;

entity df4to1mux10 is
    port (i0: in  bit_10;
          i1: in  bit_10;
          i2: in  bit_10;
          i3: in  bit_10;
          s0: in  bit;
          s1: in  bit;
          o:  out bit_10);
end df4to1mux10;


architecture df4to1mux10_a of df4to1mux10 is
begin
    o <= i0 after 0.4 ns when s1 = '0' and s0 = '0'
else
         i1 after 0.4 ns when s1 = '0' and s0 = '1'
else
         i2 after 0.4 ns when s1 = '1' and s0 = '0'
else
         i3 after 0.4 ns;
end df4to1mux10_a;
```

DF4TO1MUX16

```
library my_packages;
use my_packages.package_1.all;

entity df4to1mux16 is
    port (i0: in  bit_16;
          i1: in  bit_16;
          i2: in  bit_16;
          i3: in  bit_16;
          s0: in  bit;
          s1: in  bit;
          o:  out bit_16);
end df4to1mux16;


architecture df4to1mux16_a of df4to1mux16 is
begin
    o <= i0 after 0.4 ns when s1 = '0' and s0 = '0'
else
         i1 after 0.4 ns when s1 = '0' and s0 = '1'
else
         i2 after 0.4 ns when s1 = '1' and s0 = '0'
else
         i3 after 0.4 ns;
end df4to1mux16_a;
```

DF4TO1MUX32

```
library my_packages;
use my_packages.package_1.all;

entity df4to1mux32 is
    port(i0: in  bit_32;
         i1: in  bit_32;
         i2: in  bit_32;
         i3: in  bit_32;
         s0: in  bit;
         s1: in  bit;
         o:  out bit_32);
end df4to1mux32;


architecture df4to1mux32_a of df4to1mux32 is
begin
    o <= i0 after 0.4 ns when s1 = '0' and s0 = '0'
else
         i1 after 0.4 ns when s1 = '0' and s0 = '1'
else
         i2 after 0.4 ns when s1 = '1' and s0 = '0'
else
         i3 after 0.4 ns;
end df4to1mux32_a;
```

DF4TO1MUX4

```
library my_packages;
use my_packages.package_1.all;

entity df4to1mux4 is
    port (i0: in  bit_4;
          i1: in  bit_4;
          i2: in  bit_4;
          i3: in  bit_4;
          s0: in  bit;
          s1: in  bit;
          o:  out bit_4);
end df4to1mux4;


architecture df4to1mux4_a of df4to1mux4 is
begin
    o <= i0 after 0.4 ns when s1 = '0' and s0 = '0'
else
         i1 after 0.4 ns when s1 = '0' and s0 = '1'
else
         i2 after 0.4 ns when s1 = '1' and s0 = '0'
else
         i3 after 0.4 ns;
end df4to1mux4_a;
```

DF4TO1MUX8

```
library my_packages;
use my_packages.package_1.all;

entity df4to1mux8 is
    port (i0: in  bit_8;
          i1: in  bit_8;
          i2: in  bit_8;
          i3: in  bit_8;
          s0: in  bit;
          s1: in  bit;
          o:  out bit_8);
end df4to1mux8;


architecture df4to1mux8_a of df4to1mux8 is
begin
    o <= i0 after 0.4 ns when s1 = '0' and s0 = '0'
else
         i1 after 0.4 ns when s1 = '0' and s0 = '1'
else
         i2 after 0.4 ns when s1 = '1' and s0 = '0'
else
         i3 after 0.4 ns;
end df4to1mux8_a;
```

DFAA

```
library my_packages;
use my_packages.package_1.all;

entity dfaa is
    port(a: in  bit_32;
         b: in  bit_32;
         s: in  bit;
         o: out bit_32;
         d: out bit);
end dfaa;


architecture dfaa_a of dfaa is
begin
    o <= a + b after 7 ns when s = '1' else
         x"0000_0000";
    d <= '1' after 8 ns when s = '1' else
         '0' after 1 ns;
end dfaa_a;
```

DFADD8

```
library my_packages;
use my_packages.package_1.all;

entity dfadd8 is
    port(i:     in  bit_32;
         start: in  bit;
         o:     out bit_32;
         done:  out bit);
end dfadd8;


architecture dfadd8_a of dfadd8 is

    signal temp: bit_32;
    signal dummy, done_temp: bit;

begin

    temp <= i + x"0000_0008" after 25 ns when start =
'1' else
            temp;

    o <= temp;

    dummy <= '1' when start = '1' else
             '0' when done_temp = '1' else
             dummy;

    done_temp <= '1' after 26 ns when dummy = '1' else
                 '0';

    done <= done_temp;

end dfadd8_a;
```

DFALU

```
library my_packages;
use my_packages.package_1.all;

library df_comp;
use df_comp.dfreg32.all;
use df_comp.dfregr.all;
use df_comp.dfred.all;
use df_comp.dffed.all;
use df_comp.dfrslat.all;
use df_comp.df2to1mux32.all;
use df_comp.dfregbank.all;
use df_comp.dfalublk.all;
use df_comp.dfmd.all;
use df_comp.dfhlreg.all;
use df_comp.dfadd8.all;

entity dfalu is
    port (alu_start:   in  bit;
          ibo:         in  bit;
          alu_select:  in  bit;
          md_select:   in  bit;
          add8_select: in  bit;
          reg1_out:    in  bit_5;
          reg2_out:    in  bit_5;
          treg_out:    in  bit_5;
          pcl:         in  bit_32;
          inst_in:     in  bit_32;
          wb_data:     in  bit_32;
          wb_sel:      in  bit_6;
          vbt:         in  bit_4;
          write:       in  bit;
          jjr:         in  bit;
          set_hi_db:   in  bit;
          set_lo_db:   in  bit;
          cc:          out bit;
          hi_db:       out bit;
          lo_db:       out bit;
          db:          out bit_32;
          reg:         out bit_32;
          alu_exc:     out bit;
          inst_out:    out bit_32;
          data1_out:   out bit_32;
          data2_out:   out bit_32;
          ccd:         out bit;
          r1_test:     out bit_32;
          r2_test:     out bit_32;
          r3_test:     out bit_32;
          r4_test:     out bit_32;
          r31_test:    out bit_32;
          hi_test:     out bit_32;
          lo_test:     out bit_32;
          alu_done:    out bit);
end dfalu;


architecture dfalu_a of dfalu is

    component dfreg32
        port (d: in  bit_32;
              c: in  bit;
              q: out bit_32);
    end component;

    component dfregr
        port(d: in  bit;
             c: in  bit;
             r: in  bit;
             q: out bit);
    end component;

    component dfred
        port(i: in bit;
             o: out bit);
    end component;

    component dffed
        port(i: in bit;
             o: out bit);
    end component;

    component dfrslat
        port(s: in  bit;
             r: in  bit;
             q: out bit);
    end component;

    component df2to1mux32
        port(i0: in  bit_32;
             i1: in  bit_32;
             s:  in  bit;
             o:  out bit_32);
    end component;

    component dfregbank
        port(data:     in  bit_32;
             reg_sel:  in  bit_5;
             a_sel:    in  bit_5;
             b_sel:    in  bit_5;
             db_sel:   in  bit_5;
             write:    in  bit;
             start:    in  bit;
             vbt:      in  bit_4;
             a:        out bit_32;
             b:        out bit_32;
             db:       out bit_32;
             r1_test:  out bit_32;
             r2_test:  out bit_32;
             r3_test:  out bit_32;
             r4_test:  out bit_32;
             r31_test: out bit_32);
    end component;

    component dfalublk
        port(a_in:      in  bit_32;
             b_in:      in  bit_32;
             inst_in:   in  bit_32;
             pcl_in:    in  bit_32;
             ibo:       in  bit;
             start:     in  bit;
             lat_inst:  in  bit;
             hilo:      in  bit_32;
             add8:      in  bit_32;
             exc:       out bit;
             cc:        out bit;
             b_out:     out bit_32;
             output:    out bit_32;
             inst_out:  out bit_32;
             pcl_out:   out bit_32;
             alub_done: out bit);
    end component;

    component dfmd
        port(start: in  bit;
             x:     in  bit_32;
             y:     in  bit_32;
             sign:  in  bit;
             divd:  in  bit;
             hi:    out bit_32;
             lo:    out bit_32;
             done1: out bit;
             done2: out bit);
    end component;

    component dfhlreg
        port(start:     in  bit;
```

```
            hi_in:     in  bit_32;
            lo_in:     in  bit_32;
            load_hi:   in  bit;
            load_lo:   in  bit;
            set_hi_db: in  bit;
            set_lo_db: in  bit;
            hi_out:    out bit_32;
            lo_out:    out bit_32;
            hi_db:     out bit;
            lo_db:     out bit);
    end component;

    component dfadd8
        port(i:     in  bit_32;
             start: in  bit;
             o:     out bit_32;
             done:  out bit);
    end component;

signal a, b, hilo, add8_out, add8_in: bit_32;
signal x, y, hi1, hi2, hi3, lo1, lo2, lo3: bit_32;
signal md_inst_l: bit_32;
signal start1, exc, alub_done, alu_select_l: bit;
signal add8_select_l, add8_start, add8_done: bit;
signal alu_done_temp, sign, divd, start2: bit;
signal md_done1, md_done2, md_select_l: bit;
signal load_hi, load_lo, l_hi, l_lo: bit;
signal jjr_done, alu_start_r, alu_start_f: bit;
signal done_temp_r, done_temp: bit;
signal write_nomove: bit;

begin
    alu_start_red: dfred
        port map(alu_start, alu_start_r);

    register_bank: dfregbank
        port map(wb_data, wb_sel(4 downto 0), reg1_out,
reg2_out,
                 treg_out, write_nomove, alu_start_r,
vbt, a, b, db,
                 r1_test, r2_test, r3_test, r4_test,
r31_test);

    hi_test <= hi3;
    lo_test <= lo3;

    write_nomove <= write and (not wb_sel(5)) after 2
ns;

    reg <= a;

    alu_block: dfalublk
        port map(a, b, inst_in, pcl, ibo, start1,
alu_start_r,
                 hilo, add8_out, exc, cc, data2_out,
data1_out,
                 inst_out, add8_in, alub_done);

    alu_select_latch: dfregr
        port map(alu_select, alu_start_r,
alu_done_temp, alu_select_l);

    add8_select_latch: dfregr
        port map(add8_select, alu_start_r,
alu_done_temp, add8_select_l);

    start1 <= alu_start and alu_select_l after 1 ns;

    add8_start <= alu_start and add8_select_l after 1
ns;

    add8: dfadd8
        port map(add8_in, add8_start, add8_out,
add8_done);

    alu_exc <= exc and alu_done_temp after 1 ns;

    md: dfmd
        port map(start2, x, y, sign, divd, hi1, lo1,
md_done1,
                 md_done2);

    md_x_latch: dfreg32
        port map(a, start2, x);

    md_y_latch: dfreg32
        port map(b, start2, y);

    md_inst_latch: dfreg32
        port map(inst_in, start2, md_inst_l);

    sign <= not md_inst_l(0);

    divd <= md_inst_l(1);

    md_select_latch: dfregr
        port map(md_select, alu_start_r, alu_done_temp,
md_select_l);

    start2 <= alu_start and md_select_l after 1 ns;

    hi_mux: df2to1mux32
        port map(hi1, wb_data, wb_sel(5), hi2);

    lo_mux: df2to1mux32
        port map(lo1, wb_data, wb_sel(5), lo2);

    hilo_reg: dfhlreg
        port map(alu_start, hi2, lo2, load_hi, load_lo,
                 set_hi_db, set_lo_db, hi3, lo3, hi_db,
lo_db);

    load_hi <= l_hi or md_done2 after 1.3 ns;
    load_lo <= l_lo or md_done2 after 1.3 ns;

    l_hi <= write and wb_sel(5) and (not wb_sel(0))
after 1.4 ns;
    l_lo <= write and wb_sel(5) and      wb_sel(0)
after 1.1 ns;

    hilo_mux: df2to1mux32
        port map(hi3, lo3, md_inst_l(1), hilo);

    done_temp <= md_done1 or jjr_done or add8_done or
                 alub_done after 1.5 ns;

    done_temp_red: dfred
        port map(done_temp, done_temp_r);

    alu_start_fed: dffed
        port map(alu_start, alu_start_f);

    done_latch: dfrslat
        port map(done_temp_r, alu_start_f,
alu_done_temp);

    alu_done <= alu_done_temp;

    jjr_latch: dfregr
        port map(jjr, alu_start_r, alu_done_temp,
jjr_done);

    ccd_latch: dfrslat
        port map(done_temp_r, alu_start_r, ccd);

end dfalu_a;
```

DFALU32

```
library my_packages;
use my_packages.package_1.all;

entity dfalu32 is
    port(a:     in bit_32;
         b:     in bit_32;
         s0:    in bit;
         s1:    in bit;
         s2:    in bit;
         s3:    in bit;
         c_out: out bit;
         o:     out bit_32);
end dfalu32;


architecture dfalu32_a of dfalu32 is

    function overflow(l, r: in bit_32) return bit is
        variable carry: bit;
        variable carry30: bit;
        variable carry31: bit;
    begin
        for i in 0 to 31 loop
            if carry = '0' then
                carry := l(i) and r(i);
            else
                carry := l(i) or r(i);
            end if;
            if i = 30 then
                carry30 := carry;
            elsif i = 31 then
                carry31 := carry;
            end if;
        end loop;
```

```
        return (carry30 xor carry31);
    end overflow;

    signal at, bt, ot: bit_32;
begin

    at <= not a after 1 ns when s3 = '1' else
          a;

    bt <= not b after 1 ns when s3 = '1' else
          b;

    ot <= at - bt   after 10 ns when -- a minus b
                                  s0 = '0' and s1 = '1'
and s2 = '0' else
          at + bt   after 10 ns when -- a plus b
                                  s0 = '1' and s1 = '1'
and s2 = '0' else
          at xor bt after 10 ns when -- a xor b
                                  s0 = '0' and s1 = '0'
and s2 = '1' else
          at or bt  after 10 ns when -- a or b
                                  s0 = '1' and s1 = '0'
and s2 = '1' else
          at and bt after 10 ns when -- a and b
                                  s0 = '0' and s1 = '1'
and s2 = '1' else
          ot;

    o <= ot;

    c_out <= overflow(at, bt);

end dfalu32_a;
```

DFALUBLK

```
library my_packages;
use my_packages.package_1.all;

library df_comp;
use df_comp.dfbusctl.all;
use df_comp.dfaludec.all;
use df_comp.dfalu32.all;
use df_comp.dfovrf.all;
use df_comp.dfcomp.all;
use df_comp.dfbctl.all;
use df_comp.dfsctl.all;
use df_comp.dfshift.all;
use df_comp.dfslt.all;
use df_comp.dfoutsel.all;

entity dfalublk is
    port(a_in:      in  bit_32;
         b_in:      in  bit_32;
         inst_in:   in  bit_32;
         pcl_in:    in  bit_32;
         ibo:       in  bit;
         start:     in  bit;
         lat_inst:  in  bit;
         hilo:      in  bit_32;
         add8:      in  bit_32;
         exc:       out bit;
         cc:        out bit;
         b_out:     out bit_32;
         output:    out bit_32;
         inst_out:  out bit_32;
         pcl_out:   out bit_32;
         alub_done: out bit);
end dfalublk;


architecture dfalublk_a of dfalublk is

component dfbusctl
    port(inst_in:   in  bit_32;
         a_in:      in  bit_32;
         b_in:      in  bit_32;
         pcl_in:    in  bit_32;
         start:     in  bit;
         lat_inst:  in  bit;
         ibo:       in  bit;
         inst_out:  out bit_32;
         a_out:     out bit_32;
         b_out:     out bit_32;
         a:         out bit_5;
         b_pass:    out bit_32;
         pcl_out:   out bit_32;
         alub_done: out bit);
end component;

component dfaludec
    port(i:  in  bit_32;
         s0: out bit;
         s1: out bit;
         s2: out bit;
         s3: out bit);
end component;

component dfalu32
    port(a:     in  bit_32;
         b:     in  bit_32;
         s0:    in  bit;
         s1:    in  bit;
         s2:    in  bit;
         s3:    in  bit;
         c_out: out bit;
         o:     out bit_32);
end component;

component dfovrf
    port(inst:     in  bit_32;
         data:     in  bit_32;
         carry:    in  bit;
         overflow: out bit);
end component;

component dfcomp
    port(i:   in  bit_32;
         ltz: out bit;
         eqz: out bit;
         gtz: out bit;
         o:   out bit_32);
end component;

component dfbctl
    port(inst: in  bit_32;
         ltz:  in  bit;
         eqz:  in  bit;
         gtz:  in  bit;
         cc:   out bit);
end component;

component dfsctl
    port(inst: in  bit_32;
         a:    in  bit_5;
         lr:   out bit;
         la:   out bit;
         sel:  out bit_5);
end component;

component dfshift
    port(i:   in  bit_32;
         lr:  in  bit;
         la:  in  bit;
         sel: in  bit_5;
         o:   out bit_32);
end component;

component dfslt
    port(i:    in  bit_32;
         inst: in  bit_32;
         ltz:  in  bit;
         o:    out bit_32);
end component;

component dfoutsel
    port(inst:   in  bit_32;
         alu:    in  bit_32;
         add8:   in  bit_32;
         hilo:   in  bit_32;
         output: out bit_32);
end component;

signal instl, a_line, b_line, alu_out, comp_out:
bit_32;
signal shift_out, slt_out: bit_32;
signal a: bit_5;
signal s0, s1, s2, s3, c_out, ltz, eqz, gtz, lr, la:
bit;
signal sel: bit_5;

begin

    bus_control: dfbusctl
        port map(inst_in, a_in, b_in, pcl_in, start,
                 lat_inst, ibo, instl, a_line, b_line,
                 a, b_out, pcl_out, alub_done);

    inst_out <= instl;

    alu_decode: dfaludec
        port map(instl, s0, s1, s2, s3);
```

```
    alu: dfalu32
        port map(a_line, b_line, s0, s1,
                 s2, s3, c_out, alu_out);

    ovrf: dfovrf
        port map(instl, alu_out, c_out, exc);

    compare: dfcomp
        port map(alu_out, ltz, eqz, gtz, comp_out);

    branch_ctl: dfbctl
        port map(instl, ltz, eqz, gtz, cc);

    shift_ctl: dfsctl
        port map(instl, a, lr, la, sel);

    shifter: dfshift
        port map(comp_out, lr, la, sel, shift_out);

    slt_box: dfslt
        port map(shift_out, instl, ltz, slt_out);

    output_selector: dfoutsel
        port map(instl, slt_out, add8, hilo, output);

end dfalublk_a;
```

DFALUDEC

```
library my_packages;
use my_packages.package_1.all;

entity dfaludec is
    port(i:  in  bit_32;
         s0: out bit;
         s1: out bit;
         s2: out bit;
         s3: out bit);
end dfaludec;


architecture dfaludec_a of dfaludec is
begin

    s0 <= '1' after 2.5 ns when -- c
                                (i(31) = '0' and
                                 i(29 downto 26) =
b"1101") or
                                -- e
                                (i(29 downto 28) = b"00"
and
                                 i(26) = '0' and
                                 i(1 downto 0) = b"01")
or
                                -- f
                                (i(29 downto 28) = b"00"
and
                                 i(26) = '0' and
                                 i(2 downto 1) = b"00")
or
                                -- h
                                 i(29 downto 26) =
b"1111" or
                                -- j
                                 i(29 downto 27) = b"100"
or
                                -- m
                                (i(29 downto 28) = b"00"
and
                                 i(26) = '0' and
                                 i(5) = '0') or
                                -- n
                                 i(29 downto 27) = b"001"
or
                                -- u
                                 i(31) = '1' else
          '0' after 2.5 ns;

    s1 <= '1' after 2.5 ns when -- a
                                (i(31) = '0' and
                                 i(29 downto 28) = b"00"
and
                                 i(26) = '0' and
                                 i(5) = '1' and
                                 i(3 downto 0) = b"0111")
or
                                -- d
                                (i(29) = '1' and
                                 i(27 downto 26) = b"00")
or
                                -- h
                                 i(29 downto 26) =
b"1111" or
                                -- k
                                (i(29) = '0' and
                                 i(1 downto 0) = b"00")
or
                                -- m
                                (i(29 downto 28) = b"00"
and
                                 i(26) = '0' and
                                 i(5) = '0') or
                                -- p
                                (i(29) = '0' and
                                 i(3) = '1') or
                                -- q
                                (i(29) = '0' and
                                 i(2) = '0') or
                                -- r
                                (i(29) = '0' and
                                 i(26) = '1') or
                                -- s
                                 i(29 downto 28) = b"10"
or
                                -- t
                                 i(29 downto 28) = b"01"
or
                                -- u
                                 i(31) = '1' else
          '0' after 2.5 ns;

    s2 <= '1' after 2.5 ns when -- b
                                (i(31) = '0' and
                                 i(29 downto 28) = b"00"
and
                                 i(26) = '0' and
                                 i(5) = '1' and
                                 i(3 downto 2) = b"01")
or
                                -- c
                                (i(31) = '0' and
                                 i(29 downto 26) =
b"1101") or
                                -- g
                                (i(31) = '0' and
                                 i(29 downto 28) = b"11"
and
                                 i(26) = '0') else
          '0' after 2.5 ns;

    s3 <= '1' after 2.5 ns when -- a
                                (i(31) = '0' and
                                 i(29 downto 28) = b"00"
and
                                 i(26) = '0' and
                                 i(5) = '1' and
                                 i(3 downto 0) = b"0111")
else
          '0' after 2.5 ns;

end dfaludec_a;
```

DFASEL

```
library my_packages;
use my_packages.package_1.all;

library df_comp;
use df_comp.df4to1mux32.all;

entity dfasel is
    port(a_in:  in  bit_32;
         pcl:   in  bit_32;
         j:     in  bit;
         s:     in  bit;
         a_out: out bit_32);
end dfasel;


architecture dfasel_a of dfasel is

    component df4to1mux32
        port(i0: in  bit_32;
             i1: in  bit_32;
             i2: in  bit_32;
             i3: in  bit_32;
             s0: in  bit;
             s1: in  bit;
             o:  out bit_32);
    end component;

    constant zero32: bit_32 := x"0000_0000";
```

```
    signal zero: bit_32 := zero32;
begin

    mux: df4to1mux32
        port map(a_in, pc1, zero, zero, j, s, a_out);

end dfasel_a;
```

DFBC

```
library df_comp;
use df_comp.dfred.all;
use df_comp.dffed.all;
use df_comp.dfrslat.all;

entity dfbc is
    port (ir:  in  bit;
          il:  in  bit;
          mr:  in  bit;
          ml:  in  bit;
          ack: in  bit;
          ia:  out bit;
          ma:  out bit;
          req: out bit);
end dfbc;


architecture dfbc_a of dfbc is

    component dfred
        port(i:  in  bit;
             o: out bit);
    end component;

    component dffed
        port(i:  in  bit;
             o: out bit);
    end component;

    component dfrslat
        port(s: in  bit;
             r: in  bit;
             q: out bit);
    end component;

    signal q0, q1, clk, d: bit;
    signal iml, iml_r, ack_f: bit;
    signal ia_sel, ia_sel_r, ma_sel, ma_sel_r: bit;
    signal iml_f, q1_f: bit;
begin

    clk <= not clk after 4 ns;

    q0 <= ((not q1) and (not q0) and (not ir)) or
          ((not q1) and      q0  and      mr ) or
          (     q1  and      q0  and (not d )) or
          (     q1  and (not q0) and      d  )
          when clk'event and clk = '1' else q0;

    q1 <= ((not q1) and (not q0) and      ir ) or
          ((not q1) and      q0  and      mr ) or
          (     q1  and (not d)              )
          when clk'event and clk = '1' else q1;

    ia_sel <= q1 and (not q0) after 1 ns;

    ma_sel <= q1 and q0 after 1 ns;

    ia_sel_red: dfred
        port map(ia_sel, ia_sel_r);

    ma_sel_red: dfred
        port map(ma_sel, ma_sel_r);

    ia_latch: dfrslat
        port map(ia_sel_r, ack_f, ia);

    ma_latch: dfrslat
        port map(ma_sel_r, ack_f, ma);

    r_latch: dfrslat
        port map(iml_r, iml_f, req);

    iml_red: dfred
        port map(iml, iml_r);

    ack_fed: dffed
        port map(ack, ack_f);

    iml <= il or ml after 1 ns;

    done_latch: dfrslat
        port map(iml_f, q1_f, d);

    iml_fed: dffed
        port map(iml, iml_f);

    q1_fed: dffed
        port map(q1, q1_f);

end dfbc_a;
```

DFBCTL

```
library my_packages;
use my_packages.package_1.all;

entity dfbctl is
    port(inst: in  bit_32;
         ltz:  in  bit;
         eqz:  in  bit;
         gtz:  in  bit;
         cc:   out bit);
end dfbctl;


architecture dfbctl_a of dfbctl is

    signal cct: bit;

begin

    cct <=      eqz after 2 ns when -- beq
                                    (inst(27 downto 26)
= "00" and
                                     inst(31 downto 26)
/= "000001") else
          (not eqz) after 2 ns when -- bne
                                    (inst(27 downto 26)
= "01" and
                                     inst(31 downto 26)
/= "000001") else
       (ltz or eqz) after 2 ns when -- blez
                                    (inst(27 downto 26)
= "10" and
                                     inst(31 downto 26)
/= "000001") else
                gtz after 2 ns when -- bgtz
                                    (inst(27 downto 26)
= "11" and
                                     inst(31 downto 26)
/= "000001") else
                ltz after 2 ns when -- bltz
                                    (inst(16) = '0' and
                                     inst(31 downto 26)
= "000001") else
       (gtz or eqz) after 2 ns when -- bgez
                                     (inst(16) = '1'
and
                                     inst(31 downto 26)
= "000001") else
                cct;

    cc <= cct;

end dfbctl_a;
```

DFBJBOX

```
library my_packages;
use my_packages.package_1.all;
use my_packages.dfpack.all;

library df_comp;
use df_comp.df4to1mux4.all;
use df_comp.df4to1mux10.all;
use df_comp.df4to1mux16.all;

entity dfbjbox is
    port(inst26: in  bit_26;
         pc4:    in  bit_4;
         reg:    in  bit_30;
         b:      in  bit;
         j:      in  bit;
         jr:     in  bit;
         cc:     in  bit;
         addr30: out bit_30);
end dfbjbox;

architecture dfbjbox_a of dfbjbox is
```

```
    component df4to1mux4
        port (i0, i1, i2, i3: in  bit_4;
              s0, s1:         in  bit;
              o:              out bit_4);
    end component;

    component df4to1mux10
        port (i0, i1, i2, i3: in  bit_10;
              s0, s1:         in  bit;
              o:              out bit_10);
    end component;

    component df4to1mux16
        port (i0, i1, i2, i3: in  bit_16;
              s0, s1:         in  bit;
              o:              out bit_16);
    end component;


    constant inc_value: bit_16 := x"0001";
    constant zero4: bit_4 := x"0";
    constant zero10: bit_10 := b"00_0000_0000";

    signal z4:  bit_4  := zero4;
    signal z10: bit_10 := zero10;
    signal iv:  bit_16 := inc_value;

    signal seb:   bit;
    signal seb4:  bit_4;
    signal seb10: bit_10;

    signal m1: bit_4;
    signal m2: bit_10;
    signal m3: bit_16;

    signal s0, s1, x, y, z: bit;

begin

    seb <= inst26(15);
    seb4 <= seb & seb & seb & seb;
    seb10 <= seb & seb & seb4 & seb4;

    mux1: df4to1mux4
        port map(z4, reg(29 downto 26), pc4, seb4, s0,
s1, m1);

    mux2: df4to1mux10
        port map(z10, reg(25 downto 16), inst26(25
downto 16), seb10,
                 s0, s1, m2);

    mux3: df4to1mux16
        port map(iv, reg(15 downto 0), inst26(15 downto
0),
                 inst26(15 downto 0), s0, s1, m3);

    addr30 <= m1 & m2 & m3;

    s0 <= not (x and y) after 0.7 ns;
    s1 <= not (x and z) after 0.7 ns;

    x  <= not (cc and b) after 0.7 ns;
    y  <= not jr after 0.3 ns;
    z  <= not (y and j) after 0.7 ns;

end dfbjbox_a;
```

DFBSEL

```
library my_packages;
use my_packages.package_1.all;

library df_comp;
use df_comp.df4to1mux32.all;

entity dfbsel is
    port(inst:  in  bit_32;
         b_in:  in  bit_32;
         j:     in  bit;
         ibo:   in  bit;
         b_out: out bit_32);
end dfbsel;


architecture dfbsel_a of dfbsel is

    component df4to1mux32
        port(i0: in  bit_32;
             i1: in  bit_32;
             i2: in  bit_32;
             i3: in  bit_32;
             s0: in  bit;
             s1: in  bit;
             o:  out bit_32);
    end component;

    constant eight32: bit_32 := x"0000_0008";
    signal eight: bit_32 := eight32;

    constant zero32: bit_32 := x"0000_0000";
    signal zero: bit_32 := zero32;

    signal immed: bit_32;
    signal ext: bit_16;
    signal se4: bit_4;
    signal se16: bit_16;

    signal j_or, bcond: bit;
    signal eight_in: bit_32;

begin

    mux: df4to1mux32
        port map(b_in, immed, eight_in, eight_in, ibo,
j_or, b_out);

    immed <= ext & inst(15 downto 0);

    ext <= x"0000" when inst(28) = '1' else
           se16;

    eight_in <= zero when bcond = '1' else
                eight;

    bcond <= '1' after 1.7 ns when inst(31 downto 26) =
"000001" else
             '0' after 1.7 ns;

    j_or <= j or bcond after 1 ns;

    se16 <= se4 & se4 & se4 & se4;
    se4 <= inst(15) & inst(15) & inst(15) & inst(15);

end dfbsel_a;
```

DFBUSCTL

```
library my_packages;
use my_packages.package_1.all;

library df_comp;
use df_comp.dfreg32.all;
use df_comp.dfreg.all;
use df_comp.dfasel.all;
use df_comp.dfbsel.all;
use df_comp.dfbusdec.all;

entity dfbusctl is
    port(inst_in:   in  bit_32;
         a_in:      in  bit_32;
         b_in:      in  bit_32;
         pcl_in:    in  bit_32;
         start:     in  bit;
         lat_inst:  in  bit;
         ibo:       in  bit;
         inst_out:  out bit_32;
         a_out:     out bit_32;
         b_out:     out bit_32;
         a:         out bit_5;
         b_pass:    out bit_32;
         pcl_out:   out bit_32;
         alub_done: out bit);
end dfbusctl;


architecture dfbusctl_a of dfbusctl is

    component dfreg32
        port (d: in  bit_32;
              c: in  bit;
              q: out bit_32);
    end component;

    component dfreg
        port(d, c: in  bit;
             q:    out bit);
    end component;

    component dfasel
        port(a_in:  in  bit_32;
             pcl:   in  bit_32;
```

```
            j:     in  bit;
            s:     in  bit;
            a_out: out bit_32);
    end component;

    component dfbsel
        port(inst:  in  bit_32;
             b_in:  in  bit_32;
             j:     in  bit;
             ibo:   in  bit;
             b_out: out bit_32);
    end component;

    component dfbusdec
        port(inst: in  bit_32;
             j:    out bit;
             s:    out bit);
    end component;

    signal a_lat, b_lat, pcl_lat, inst_lat: bit_32;
    signal immed, j, s: bit;
    signal dummy, done_temp: bit;
begin

    a_in_latch: dfreg32
        port map(a_in, start, a_lat);

    pcl_latch: dfreg32
        port map(pcl_in, start, pcl_lat);

    pcl_out <= pcl_lat;

    b_in_latch: dfreg32
        port map(b_in, start, b_lat);

    inst_latch: dfreg32
        port map(inst_in, lat_inst, inst_lat);

    ibo_latch: dfreg
        port map(ibo, start, immed);

    a <= a_lat(4 downto 0);

    b_pass <= b_lat;

    a_bus_selector: dfasel
        port map(a_lat, pcl_lat, j, s, a_out);

    b_bus_selector: dfbsel
        port map(inst_lat, b_lat, j, immed, b_out);

    bus_select_decoder: dfbusdec
        port map(inst_lat, j, s);

    inst_out <= inst_lat;

    dummy <= '1' when start = '1' else
             '0' when done_temp = '1' else
             dummy;

    done_temp <= '1' after 30 ns when dummy = '1' else
                 '0' after 1 ns;

    alub_done <= done_temp;

end dfbusctl_a;
```

DFBUSDEC

```
library my_packages;
use my_packages.package_1.all;

entity dfbusdec is
    port(inst: in  bit_32;
         j:    out bit;
         s:    out bit);
end dfbusdec;


architecture dfbusdec_a of dfbusdec is
begin
    j <= '1' after 3.7 ns when -- jal or jalr
                               (inst(31 downto 26) =
"000011") or
                               (inst(31 downto 26) =
"000000" and
                                inst(5 downto 0) =
"001001") else
         '0' after 3.7 ns;

    s <= '1' after 3.6 ns when -- sXXv
                                (inst(31 downto 26) =
"000000" and
                                 inst(5 downto 2) =
"0001") else
         '0' after 3.6 ns;

end dfbusdec_a;
```

DFCOMP

```
library my_packages;
use my_packages.package_1.all;

entity dfcomp is
    port(i:   in  bit_32;
         ltz: out bit;
         eqz: out bit;
         gtz: out bit;
         o:   out bit_32);
end dfcomp;


architecture dfcomp_a of dfcomp is
begin
    o <= i;

    ltz <= i(31);

    gtz <= '1' after 4.4 ns when i(31) = '0' and i /=
x"0000_0000" else
           '0';

    eqz <= '1' after 3.1 ns when i = x"0000_0000" else
           '0';

end dfcomp_a;
```

DFCOMPARE

```
library my_packages;
use my_packages.package_1.all;

use std.textio.all;

entity dfcompare is
    port(compare:          in  question_type;
         compare_load:     in  question_type;
         pc_test:          in  bit_32;
         r1_test:          in  bit_32;
         r2_test:          in  bit_32;
         r3_test:          in  bit_32;
         r4_test:          in  bit_32;
         r31_test:         in  bit_32;
         hi_test:          in  bit_32;
         lo_test:          in  bit_32;
         compare_ack:      out question_type;
         compare_load_ack: out question_type);
end dfcompare;


architecture dfcompare_a of dfcompare is
begin

    compare_block: process
        type test_field_type is array (0 to 9) of
bit_32;
        type test_array_type is array (0 to 1000) of
test_field_type;
        variable expect: test_array_type;
        variable index: integer range 0 to 1000;

        file in1: text is in "expected";
        variable line1: line;
        variable inst: bit_32;
        variable skip: question_type;
    begin
        wait on compare, compare_load;

        if skip = no then
            if compare_load = yes then
                index := 0;
                while endfile(in1) = false loop
                    for i in 0 to 9 loop
                        if endfile(in1) = false then
                            readline(in1, line1);
                            read(line1, inst);
                            expect(index)(i) := inst;
                        end if;
                    end loop;
```

```
                index := index + 1;
            end loop;
            index := 0;
            skip := yes;
            compare_load_ack <= yes after delay;
        end if;
    else
        if compare = yes then
            assert pc_test = expect(index)(0)
                report "PC RRGISTER ERROR"
                severity warning;
            assert r1_test = expect(index)(1)
                report "REGISTER 1 ERROR"
                severity warning;
            assert r2_test = expect(index)(2)
                report "REGISTER 2 ERROR"
                severity warning;
            assert r3_test = expect(index)(3)
                report "REGISTER 3 ERROR"
                severity warning;
            assert r4_test = expect(index)(4)
                report "REGISTER 4 ERROR"
                severity warning;
            assert r31_test = expect(index)(5)
                report "REGISTER 31 ERROR"
                severity warning;
            assert hi_test = expect(index)(6)
                report "REGISTER HI ERROR"
                severity warning;
            assert lo_test = expect(index)(7)
                report "REGISTER LO ERROR"
                severity warning;
            index := index + 1;
            compare_ack <= yes after delay;
        else
            compare_ack <= no after delay;
        end if;  -- if compare...
    end if;  -- if skip...
end process compare_block;

end dfcompare_a;
```

DFCPU

```
library my_packages;
use my_packages.package_1.all;

library df_comp;
use df_comp.dfhcc.all;
use df_comp.dfif.all;
use df_comp.dfid.all;
use df_comp.dfalu.all;
use df_comp.dfmem.all;
use df_comp.dfwb.all;
use df_comp.dfeh.all;
use df_comp.dfbc.all;
use df_comp.dfrslat.all;
use df_comp.dfreg32.all;

entity dfcpu is
    port(sys_control_sig:  in    sys_control_type;
         memory_ack:       in    bit;
         memory_load_ack:  in    question_type;
         compare_ack:      in    question_type;
         compare_load_ack: in    question_type;
         addr_bus:         inout bus_bit_32 bus;
         data_bus:         inout bus_bit_32 bus;
         memory_req:       out   bit;
         memory_w:         out   bit;
         memory_opcode:    out   bit_3;
         memory_load:      out   question_type;
         compare:          out   question_type;
         compare_load:     out   question_type;
         pc_test:          out   bit_32;
         r1_test:          out   bit_32;
         r2_test:          out   bit_32;
         r3_test:          out   bit_32;
         r4_test:          out   bit_32;
         r31_test:         out   bit_32;
         hi_test:          out   bit_32;
         lo_test:          out   bit_32);
end dfcpu;


architecture dfcpu_a of dfcpu is

    -- component declarations
    component dfhcc
        port (init:    in  bit;
              prev_ok: in  bit;
              ready:   in  bit;
              ain:     in  bit;
              aout:    out bit;
              rout:    out bit;
              ok:      out bit);
    end component;

    component dfif
        port (if_start:   in    bit;
              ia:         in    bit;
              int_req:    in    bit;
              addr_valid: in    bit;
              iv:         in    bit_32;
              new_pc:     in    bit_32;
              addr_bus:   inout bus_bit_32;
              data_bus:   inout bus_bit_32;
              ir:         out   bit;
              il:         out   bit;
              inst:       out   bit_32;
              pc:         out   bit_32;
              if_exc:     out   bit;
              if_done:    out   bit);
    end component;

    component dfid
        port (id_start:    in  bit;
              hi_db:       in  bit;
              lo_db:       in  bit;
              db:          in  bit_32;
              inst_in:     in  bit_32;
              cc:          in  bit;
              ccd:         in  bit;
              pc:          in  bit_32;
              ir:          in  bit;
              reg:         in  bit_32;
              illegal:     out bit;
              id_exc:      out bit;
              ibo:         out bit;
              alu_select:  out bit;
              md_select:   out bit;
              add8_select: out bit;
              reg1_out:    out bit_5;
              reg2_out:    out bit_5;
              treg_out:    out bit_5;
              addr_valid:  out bit;
              new_pc:      out bit_32;
              pcl:         out bit_32;
              inst_out:    out bit_32;
              set_hi_db:   out bit;
              set_lo_db:   out bit;
              jjr:         out bit;
              id_done:     out bit);
    end component;

    component dfalu
        port (alu_start:   in  bit;
              ibo:         in  bit;
              alu_select:  in  bit;
              md_select:   in  bit;
              add8_select: in  bit;
              reg1_out:    in  bit_5;
              reg2_out:    in  bit_5;
              treg_out:    in  bit_5;
              pcl:         in  bit_32;
              inst_in:     in  bit_32;
              wb_data:     in  bit_32;
              wb_sel:      in  bit_6;
              vbt:         in  bit_4;
              write:       in  bit;
              jjr:         in  bit;
              set_hi_db:   in  bit;
              set_lo_db:   in  bit;
              cc:          out bit;
              hi_db:       out bit;
              lo_db:       out bit;
              db:          out bit_32;
              reg:         out bit_32;
              alu_exc:     out bit;
              inst_out:    out bit_32;
              data1_out:   out bit_32;
              data2_out:   out bit_32;
              ccd:         out bit;
              r1_test:     out bit_32;
              r2_test:     out bit_32;
              r3_test:     out bit_32;
              r4_test:     out bit_32;
              r31_test:    out bit_32;
              hi_test:     out bit_32;
              lo_test:     out bit_32;
              alu_done:    out bit);
    end component;

    component dfmem
        port (mem_start: in    bit;
```

```
            ma:         in    bit;
            inst_in:    in    bit_32;
            data1_in:   in    bit_32;
            data2_in:   in    bit_32;
            addr_bus:   inout bus_bit_32;
            data_bus:   inout bus_bit_32;
            mem_exc:    out   bit;
            mr:         out   bit;
            ml:         out   bit;
            w:          out   bit;
            opcode:     out   bit_3;
            inst_out:   out   bit_32;
            data_out:   out   bit_32;
            vbt:        out   bit_4;
            mem_done:   out   bit);
    end component;

    component dfwb
        port (wb_start: in  bit;
              inst_in:  in  bit_32;
              data_in:  in  bit_32;
              vbt_in:   in  bit_4;
              wb_data:  out bit_32;
              wb_sel:   out bit_6;
              vbt:      out bit_4;
              wb_done:  out bit);
    end component;

    component dfeh
        port (illegal:    in  bit;
              if_exc:     in  bit;
              id_exc:     in  bit;
              alu_exc:    in  bit;
              mem_exc:    in  bit;
              int_req:    out bit;
              int_vector: out bit_32);
    end component;

    component dfbc
        port (ir:  in  bit;
              il:  in  bit;
              mr:  in  bit;
              ml:  in  bit;
              ack: in  bit;
              ia:  out bit;
              ma:  out bit;
              req: out bit);
    end component;

    component dfrslat
        port(s: in  bit;
             r: in  bit;
             q: out bit);
    end component;

    component dfreg32
        port (d: in  bit_32;
              c: in  bit;
              q: out bit_32);
    end component;


    -- system control
    signal start: bit;

    signal begin_in: bit;

    -- pipeline handshake
    signal begin_ok, if_ok, id_ok, alu_ok, mem_ok,
wb_ok: bit := '1';
    signal if_ack, id_ack, alu_ack, mem_ack, wb_ack,
end_ack: bit;
    signal if_start, id_start, alu_start, mem_start,
wb_start: bit;
    signal if_done, id_done, alu_done, mem_done,
wb_done: bit;
    signal init: bit;

    -- interrupt/exception control
    signal if_exc, id_exc, alu_exc, mem_exc: bit;
    signal illegal, int_req: bit;
    signal int_vector: bit_32;
    signal int_latch_q: bit;

    constant zero_constant: bit := '0';
    signal int_reset: bit := zero_constant;

    -- bus control
    signal if_bus_req, mem_bus_req: bit;
    signal if_bus_ack, mem_bus_ack: bit;
    signal if_load_addr, mem_load_addr: bit;

    -- data, control
    signal inst_ifid, pc, new_pc: bit_32;
    signal addr_valid: bit;

    signal cc, hi_db, lo_db: bit;
    signal db, reg: bit_32;
    signal ibo, alu_select, md_select, add8_select:
 bit;
    signal reg1_out, reg2_out, treg_out: bit_5;
    signal pcl, inst_idalu: bit_32;
    signal set_hi_db, set_lo_db, jjr: bit;

    signal wb_data, data1_out, data2_out, inst_alumem:
 bit_32;
    signal wb_sel: bit_6;
    signal vbt: bit_4;

    signal inst_memwb, data_memwb: bit_32;
    signal vbt_memwb: bit_4;

    signal ccd: bit;

    -- signals for test bench
    signal pc_alumem, pc_memwb: bit_32;
    signal test_mode, wb_done2: bit;
 begin

    system: process
    begin
        wait on sys_control_sig;
        case sys_control_sig is
          when stop =>
            start <= '0';

          when reset =>
            null;

          when load =>
            memory_load <= yes after delay;
            wait until memory_load_ack = yes;

            if test_mode = '1' then
                compare_load <= yes after delay;
                wait until compare_load_ack = yes;
            end if;

          when run =>
            start <= '1';
        end case;
    end process system;

    int_latch: dfrslat
        port map(int_req, int_reset, int_latch_q);

    bus_control: dfbc
        port map(if_bus_req, if_load_addr, mem_bus_req,
mem_load_addr,
                 memory_ack, if_bus_ack, mem_bus_ack,
memory_req);

    exception_handler: dfeh
        port map(illegal, if_exc, id_exc, alu_exc,
                 mem_exc, int_req, int_vector);

    begin_in <= not (if_ack or start) after 1 ns;

    begin_ok <= begin_in or int_latch_q after 1.3 ns;

    if_hcc: dfhcc
        port map(init, begin_ok, if_done, id_ack,
                 if_ack, if_start, if_ok);

    id_hcc: dfhcc
        port map(init, if_ok, id_done, alu_ack,
                 id_ack, id_start, id_ok);

    alu_hcc: dfhcc
        port map(init, id_ok, alu_done, mem_ack,
                 alu_ack, alu_start, alu_ok);

    mem_hcc: dfhcc
        port map(init, alu_ok, mem_done, wb_ack,
                 mem_ack, mem_start, mem_ok);

    wb_hcc: dfhcc
        port map(init, mem_ok, wb_done2, end_ack,
                 wb_ack, wb_start, wb_ok);

    end_ack  <= not wb_ok after 0.3 ns;


    if_stage: dfif
        port map(if_start, if_bus_ack, int_req,
addr_valid,
```

```
                int_vector, new_pc, addr_bus,
data_bus,if_bus_req,
                if_load_addr, inst_ifid, pc, if_exc,
if_done);

    id_stage: dfid
        port map(id_start, hi_db, lo_db, db, inst_ifid,
cc, ccd,
                pc, if_bus_req, reg, illegal, id_exc,
ibo, alu_select,
                md_select, add8_select, reg1_out,
reg2_out, treg_out,
                addr_valid, new_pc, pcl, inst_idalu,
set_hi_db,
                set_lo_db, jjr, id_done);

    alu_stage: dfalu
        port map(alu_start, ibo, alu_select, md_select,
add8_select,
                reg1_out, reg2_out, treg_out, pcl,
inst_idalu, wb_data,
                wb_sel, vbt, wb_done, jjr, set_hi_db,
set_lo_db,
                cc, hi_db, lo_db, db, reg, alu_exc,
inst_alumem,
                data1_out, data2_out, ccd, r1_test,
r2_test, r3_test,
                r4_test, r31_test, hi_test, lo_test,
alu_done);

    mem_stage: dfmem
        port map(mem_start, mem_bus_ack, inst_alumem,
data1_out,
                data2_out, addr_bus, data_bus,
mem_exc, mem_bus_req,
                mem_load_addr, memory_w,
memory_opcode, inst_memwb,
                data_memwb, vbt_memwb, mem_done);

    wb_stage: dfwb
        port map(wb_start, inst_memwb, data_memwb,
vbt_memwb,
                wb_data, wb_sel, vbt, wb_done);

    alu_pc_latch: dfreg32
        port map(pcl, alu_start, pc_alumem);

    mem_pc_latch: dfreg32
        port map(pc_alumem, mem_start, pc_memwb);

    wb_pc_latch: dfreg32
        port map(pc_memwb, wb_start, pc_test);

    hcc_tb_control: process
    begin
        wait on wb_done;

        if test_mode = '0' then
            if wb_done = '1' then
                wb_done2 <= '1';
            else
                wb_done2 <= '0';
            end if;
        else
            if wb_done = '1' then
                compare <= yes after delay;
                wait until compare_ack = yes;
                compare <= no after 1 ns;
                wait until compare_ack = no;
                wb_done2 <= '1';
            else
                wb_done2 <= '0';
            end if;
        end if;
    end process hcc_tb_control;

end dfcpu_a;
```

DFDBOX

```
library my_packages;
use my_packages.package_1.all;

library df_comp;
use df_comp.dftrds.all;
use df_comp.df32to1mux.all;

entity dfdbox is
    port (inst:      in  bit_32;
          db:        in  bit_32;
          hi_db:     in  bit;
          lo_db:     in  bit;
          ct_start:  in  bit;
          ts1:       in  bit;
          ts2:       in  bit;
          ttar:      in  bit;
          thi:       in  bit;
          tlo:       in  bit;
          trs0:      in  bit;
          trs1:      in  bit;
          clean:     out bit;
          dirty_reg: out bit_5);
end dfdbox;


architecture dfdbox_a of dfdbox is

    component dftrds
        port(inst: in  bit_32;
             trs0: in  bit;
             trs1: in  bit;
             reg:  out bit_5);
    end component;

    component df32to1mux
        port(i: in bit_32;
             s: in bit_5;
             o: out bit);
    end component;

    signal a: bit := '1';
    signal b, d, e, f, g, h: bit;
    signal n, p, q: bit;
    signel reg: bit_5;
    signal c: bit;

begin

    clean <= not (a or b or c) after 1.1 ns;
    dirty_reg <= reg;

    c <= not ct_start after 0.3 ns;

    a <= not (d and e) after 0.7 ns;
    b <= not (f and g and h) after 0.8 ns;
    d <= not (hi_db and thi) after 0.7 ns;
    e <= not (lo_db and tlo) after 0.7 ns;
    f <= not (ts1 and n) after 0.7 ns;
    g <= not (ts2 and p) after 0.7 ns;
    h <= not (ttar and q) after 0.7 ns;

    mux1: df32to1mux
        port map (db, inst(25 downto 21), n);

    mux2: df32to1mux
        port map (db, inst(20 downto 16), p);

    mux3: df32to1mux
        port map (db, reg, q);

    trds: dftrds
        port map (inst, trs0, trs1, reg);

end dfdbox_a;
```

DFEH

```
library my_packages;
use my_packages.package_1.all;

entity dfeh is
    port (illegal:    in  bit;
          if_exc:     in  bit;
          id_exc:     in  bit;
          alu_exc:    in  bit;
          mem_exc:    in  bit;
          int_req:    out bit;
          int_vector: out bit_32);
end dfeh;


architecture dfeh_a of dfeh is
begin

    int_req <= illegal or if_exc or id_exc or
               alu_exc or mem_exc after 1.5 ns;

    int_vector <= x"0000_fff0";

end dfeh_a;
```

DFFED

```
entity dffed is
    port(i: in bit;
         o: out bit);
end dffed;


architecture dffed_a of dffed is
    signal temp: bit;
begin
    temp <= '1' after 0.4 ns when i'event and i = '0'
else
            '0' after 1.2 ns when temp = '1' else
            temp;

    o <= temp;

end dffed_a;
```

DFHCC

```
entity dfhcc is
    port (init:    in  bit;
          prev_ok: in  bit;
          ready:   in  bit;
          ain:     in  bit;
          aout:    out bit;
          rout:    out bit;
          ok:      out bit);
end dfhcc;


architecture dfhcc_a of dfhcc is
    signal aout_temp, rout_temp, ok_temp: bit;
begin
    aout_temp <= '1' after 2.2 ns when prev_ok = '0'
and
                                       rout_temp = '0'
and
                                       ready = '0' else
                 '0' after 2.2 ns when prev_ok = '1'
and
                                       rout_temp = '1'
else
                 aout_temp;
    aout <= aout_temp;
    rout_temp <= '1' after 2.5 ns when aout_temp = '1'
and
                                       ain = '0' else
                 '0' after 1.3 ns when ain = '1' else
                 rout_temp;
    rout <= rout_temp;
    ok_temp <= '1' after 1 ns when rout_temp = '0' else
               '0' after 1 ns when rout_temp = '1' and
                                   ready = '1' else
               ok_temp;
    ok <= ok_temp;
end dfhcc_a;
```

DFHLREG

```
library my_packages;
use my_packages.package_1.all;

entity dfhlreg is
    port(start:     in  bit;
         hi_in:     in  bit_32;
         lo_in:     in  bit_32;
         load_hi:   in  bit;
         load_lo:   in  bit;
         set_hi_db: in  bit;
         set_lo_db: in  bit;
         hi_out:    out bit_32;
         lo_out:    out bit_32;
         hi_db:     out bit;
         lo_db:     out bit);
end dfhlreg;


architecture dfhlreg_a of dfhlreg is
    signal hi_reg, lo_reg: bit_32;
    signal hi_db1, hi_db2: bit;
    signal lo_db1, lo_db2: bit;
begin
    hi_reg <= hi_in after 0.3 ns when load_hi'event and
                                      load_hi = '1' else
              hi_reg;

    lo_reg <= lo_in after 0.3 ns when load_lo'event and
                                      load_lo = '1' else
              lo_reg;

    hi_out <= hi_reg after 0.6 ns;
    lo_out <= lo_reg after 0.6 ns;


    hi_db1 <= hi_db1 xor '1' after 1 ns when
                                       start'event and
                                       start = '1' and
                                       set_hi_db = '1'
else
              hi_db1;

    lo_db1 <= lo_db1 xor '1' after 1 ns when
                                       start'event and
                                       start = '1' and
                                       set_lo_db = '1'
else
              lo_db1;

    hi_db2 <= hi_db2 xor '1' after 1 ns when
                                       load_hi'event
and
                                       load_hi = '1'
else
              hi_db2;

    lo_db2 <= lo_db2 xor '1' after 1 ns when
                                       load_lo'event
and
                                       load_lo = '1'
else
              lo_db2;

    hi_db <= hi_db1 xor hi_db2 after 1 ns;
    lo_db <= lo_db1 xor lo_db2 after 1 ns;

end dfhlreg_a;
```

DFID

```
library my_packages;
use my_packages.package_1.all;

library df_comp;
use df_comp.dfrslat.all;
use df_comp.dfred.all;
use df_comp.dffed.all;
use df_comp.dfreg32.all;
use df_comp.dfreg5.all;
use df_comp.dfreg.all;
use df_comp.dfaa.all;
use df_comp.dfinstdec.all;
use df_comp.df32to1mux.all;
use df_comp.df2to1mux30.all;
use df_comp.dfbjbox.all;
use df_comp.dfdbox.all;

entity dfid is
    port (id_start:    in  bit;
          hi_db:       in  bit;
          lo_db:       in  bit;
          db:          in  bit_32;
          inst_in:     in  bit_32;
          cc:          in  bit;
          ccd:         in  bit;
          pc:          in  bit_32;
          ir:          in  bit;
          reg:         in  bit_32;
          illegal:     out bit;
          id_exc:      out bit;
          ibo:         out bit;
          alu_select:  out bit;
          md_select:   out bit;
          add8_select: out bit;
          reg1_out:    out bit_5;
          reg2_out:    out bit_5;
          treg_out:    out bit_5;
          addr_valid:  out bit;
          new_pc:      out bit_32;
          pc1:         out bit_32;
          inst_out:    out bit_32;
          set_hi_db:   out bit;
          set_lo_db:   out bit;
          jjr:         out bit;
          id_done:     out bit);
end dfid;
```

```
architecture dfid_a of dfid is

    component dfred
        port(i: in  bit;
             o: out bit);
    end component;

    component dffed
        port(i:  in  bit;
             o:  out bit);
    end component;

    component dfrslat
        port(s: in  bit;
             r: in  bit;
             q: out bit);
    end component;

    component dfreg32
        port (d: in  bit_32;
              c: in  bit;
              q: out bit_32);
    end component;

    component dfreg5
        port (d: in  bit_5;
              c: in  bit;
              q: out bit_5);
    end component;

    component dfreg
        port(d: in  bit;
             c: in  bit;
             q: out bit);
    end component;

    component dfaa
        port(a: in  bit_32;
             b: in  bit_32;
             s: in  bit;
             o: out bit_32;
             d: out bit);
    end component;

    component dfinstdec
        port(i:    in  bit_32;
             md:   out bit;
             alu:  out bit;
             ibo:  out bit;
             b:    out bit;
             j:    out bit;
             ill:  out bit;
             ts1:  out bit;
             ts2:  out bit;
             ttar: out bit;
             thi:  out bit;
             tlo:  out bit;
             exc:  out bit;
             r:    out bit;
             l:    out bit;
             trs0: out bit;
             trs1: out bit;
             add8: out bit);
    end component;

    component df32to1mux
        port(i: in  bit_32;
             s: in  bit_5;
             o: out bit);
    end component;

    component df2to1mux30
        port(i0: in  bit_30;
             i1: in  bit_30;
             s:  in  bit;
             o:  out bit_30);
    end component;

    component dfbjbox
        port(inst26: in  bit_26;
             pc4:    in  bit_4;
             reg:    in  bit_30;
             b:      in  bit;
             j:      in  bit;
             jr:     in  bit;
             cc:     in  bit;
             addr30: out bit_30);
    end component;

    component dfdbox
        port (inst:      in  bit_32;
              db:        in  bit_32;
              hi_db:     in  bit;
              lo_db:     in  bit;
              ct_start:  in  bit;
              ts1:       in  bit;
              ts2:       in  bit;
              ttar:      in  bit;
              thi:       in  bit;
              tlo:       in  bit;
              trs0:      in  bit;
              trs1:      in  bit;
              clean:     out bit;
              dirty_reg: out bit_5);
    end component;


    signal instl: bit_32;
    signal b, j, ttar, ts1, ts2, thi, tlo, r, l, trs0,
trs1: bit;
    signal last_inst: bit_32;
    signal ct_start, clean: bit;
    signal id_done_out, id_done_r: bit;
    signal id_start_r, id_start_f: bit;
    signal av, av_f: bit;
    signal aas, aad, aad_r: bit;
    signal ir_f: bit;
    signal b_sel, j_sel, r_sel: bit;
    signal bi, a, c: bit;
    signal addr30, aa_mux_out: bit_30;
    signal aa_a_input, aa_b_input, aa_out: bit_32;
    signal pc_1: bit_32;
    signal reg1, reg2: bit_5;
    signal aas_r, aas_1: bit;

begin

    inst_decoder: dfinstdec
        port map(instl, md_select, alu_select, ibo, b,
j, illegal,
                 ts1, ts2, ttar, thi, tlo, id_exc, r,
l, trs0, trs1,
                 add8_select);

    dirty_box: dfdbox
        port map(instl, db, hi_db, lo_db, ct_start,
                 ts1, ts2, ttar, thi, tlo,
                 trs0, trs1, clean, treg_out);

    reg1 <= instl(25 downto 21);
    reg2 <= instl(20 downto 16);
    inst_out <= instl;

    reg1_latch: dfreg5
        port map(reg1, id_done_r, reg1_out);

    reg2_latch: dfreg5
        port map(reg2, id_done_r, reg2_out);

    ct_latch: dfrslat
        port map(av_f, id_done_r, ct_start);

    id_done_latch: dfrslat
        port map(clean, id_start_f, id_done_out);

    id_done <= id_done_out;

    id_done_red: dfred
        port map(id_done_out, id_done_r);

    av_latch: dfrslat
        port map(ir_f, aad_r, av);

    addr_valid <= av;

    av_fed: dffed
        port map(av, av_f);

    ir_fed: dffed
        port map(ir, ir_f);

    id_start_fed: dffed
        port map(id_start, id_start_f);

    id_start_red: dfred
        port map(id_start, id_start_r);

    branch_latch: dfreg
        port map(b, ir_f, b_sel);

    jump_latch: dfreg
        port map(j, ir_f, j_sel);

    jump_reg_latch: dfreg
        port map(r, ir_f, r_sel);
```

```
    inst_latch: dfreg32
        port map(inst_in, id_start_r, instl);

    last_inst_latch: dfreg32
        port map(instl, ir_f, last_inst);

    aas <= not (a or c) after 1 ns;
    bi <= not b_sel after 0.3 ns;
    a <= not (bi or ccd) after 1 ns;
    c <= not id_start after 0.3 ns;

    aad_red: dfred
        port map(aad, aad_r);

    aa_b_input <= addr30 & b"00";
    aa_a_input <= pc_l(31 downto 2) & b"00";

    pc_latch: dfreg32
        port map(pc, id_start_r, pc_l);

    pcl <= pc_l;

    aas_red: dfred
        port map(aas, aas_r);

    aas_latch: dfrslat
        port map(aas_r, id_done_r, aas_l);

    address_adder: dfaa
        port map(aa_a_input, aa_b_input, aas_l, aa_out,
aad);

    aa_mux: df2to1mux30
        port map(aa_out(31 downto 2), addr30, j_sel,
aa_mux_out);

    new_pc <= aa_mux_out & b"00";

    bjbox: dfbjbox
        port map(last_inst(25 downto 0), pc_l(31 downto
28),
                 reg(31 downto 2), b_sel, j_sel,
                 r_sel, cc, addr30);

    jjr <= '1' after 1.3 ns when j = '1' and l = '0'
else
           '0' after 1.3 ns;

    set_lo_db <= '1' after 1.3 ns when tlo = '1' and
ttar = '0' else
                 '0' after 1.3 ns;

    set_hi_db <= '1' after 1.3 ns when thi = '1' and
ttar = '0' else
                 '0' after 1.3 ns;

end dfid_a;
```

DFIF

```
library my_packages;
use my_packages.package_1.all;

library df_comp;
use df_comp.dfred.all;
use df_comp.dffed.all;
use df_comp.dfrslat.all;
use df_comp.dfreg32.all;
use df_comp.dftsb32.all;
use df_comp.df2to1mux32.all;

entity dfif is
    port (if_start:   in    bit;
          ia:         in    bit;
          int_req:    in    bit;
          addr_valid: in    bit;
          iv:         in    bit_32;
          new_pc:     in    bit_32;
          addr_bus:   inout bus_bit_32;
          data_bus:   inout bus_bit_32;
          ir:         out   bit;
          il:         out   bit;
          inst:       out   bit_32;
          pc:         out   bit_32;
          if_exc:     out   bit;
          if_done:    out   bit);
end dfif;

architecture dfif_a of dfif is

    component dfred
        port(i: in  bit;
             o: out bit);
    end component;

    component dffed
        port(i:  in  bit;
             o:  out bit);
    end component;

    component dfrslat
        port(s: in  bit;
             r: in  bit;
             q: out bit);
    end component;

    component dfreg32
        port (d: in  bit_32;
              c: in  bit;
              q: out bit_32);
    end component;

    component dftsb32
        port(i: in  bit_32;
             e: in  bit;
             o: out bit_32);
    end component;

    component df2to1mux32
        port(i0: in  bit_32;
             i1: in  bit_32;
             s:  in  bit;
             o:  out bit_32);
    end component;

    signal start_r, ia_f, a, b, c, start: bit;
    signal ia_r, ilq: bit;
    signal ilq_f, if_start_f: bit;
    signal mux_out, mux_latch_out: bit_32;

begin

    ir_latch: dfrslat
        port map(start_r, ia_f, ir);

    start <= not (a or c) after 1 ns;
    a     <= not (int_req or b) after 1 ns;
    b     <= not addr_valid after 0.3 ns;
    c     <= not if_start after 0.3 ns;

    start_red: dfred
        port map(start, start_r);

    ia_fed: dffed
        port map(ia, ia_f);

    il_latch: dfrslat
        port map(ia_r, ia_f, ilq);

    ia_red: dfred
        port map(ia, ia_r);

    il <= ilq;

    done_latch: dfrslat
        port map(ilq_f, if_start_f, if_done);

    ilq_fed: dffed
        port map(ilq, ilq_f);

    if_start_fed: dffed
        port map(if_start, if_start_f);

    addr_mux: df2to1mux32
        port map(new_pc, iv, int_req, mux_out);

    mux_latch: dfreg32
        port map(mux_out, start_r, mux_latch_out);

    pc <= mux_latch_out;

    tsb: dftsb32
        port map(mux_latch_out, ia, addr_bus);

    db_latch: dfreg32
        port map(data_bus, ia_f, inst);

    if_exc <= ia and (mux_latch_out(0) or
mux_latch_out(1)) after 2.6 ns;

end dfif_a;
```

DFINSTDEC

```
library my_packages;
use my_packages.package_1.all;

entity dfinstdec is
    port(i:    in  bit_32;
         md:   out bit;
         alu:  out bit;
         ibo:  out bit;
         b:    out bit;
         j:    out bit;
         ill:  out bit;
         ts1:  out bit;
         ts2:  out bit;
         ttar: out bit;
         thi:  out bit;
         tlo:  out bit;
         exc:  out bit;
         r:    out bit;
         l:    out bit;
         trs0: out bit;
         trs1: out bit;
         add8: out bit);
end dfinstdec;


architecture dfinstdec_a of dfinstdec is
begin

    md   <= '1' after 4.4 ns when -- a15
                                  (i(31) = '0' and
                                   i(29 downto 26) =
"0000" and
                                   i(4 downto 3) = "11")
else
            '0' after 4.4 ns;

    alu  <= '1' after 4.4 ns when -- a2
                                  (i(31) = '0' and
                                   i(29 downto 26) =
"0000" and
                                   i(5 downto 2) =
"0010" and
                                   i(0) = '1') or
                                  -- a13
                                  (i(31) = '0' and
                                   i(28 downto 26) =
"000" and
                                   i(3) = '0' and
                                   i(0) = '0') or
                                  -- a14
                                  (i(31) = '0' and
                                   i(29 downto 26) =
"0000" and
                                   i(4 downto 3) = "00")
or
                                  -- a20
                                  (i(27) = '0' and
                                   i(4 downto 3) = "10"
and
                                   i(0) = '1') or
                                  -- a21
                                  (i(31) = '0' and
                                   i(29 downto 26) =
"0011") or
                                  -- a25
                                  (i(31) = '0' and
                                   i(29 downto 26) =
"0000" and
                                   i(5) = '1') or
                                  -- a26
                                  (i(31) = '1' and
                                   i(29) = '1') or
                                  -- a27
                                  (i(31) = '0' and
                                   i(29) = '0' and
                                   i(27 downto 26) =
"01") or
                                  -- a29
                                  (i(31) = '0' and
                                   i(29 downto 28) =
"01") or
                                  -- a30
                                  (i(31) = '1' and
                                   i(29) = '0') or
                                  -- a31
                                  (i(31) = '0' and
                                   i(29) = '1') else
            '0' after 4.4 ns;

    ibo  <= '1' after 4.4 ns when -- a26
                                  (i(31) = '1' and
                                   i(29) = '1') or
                                  -- a30
                                  (i(31) = '1' and
                                   i(29) = '0') or
                                  -- a31
                                  (i(31) = '0' and
                                   i(29) = '1') else
            '0' after 4.4 ns;

    b    <= '1' after 4.4 ns when -- a27
                                  (i(31) = '0' and
                                   i(29) = '0' and
                                   i(27 downto 26) =
"01") or
                                  -- a29
                                  (i(31) = '0' and
                                   i(29 downto 28) =
"01") else
            '0' after 4.4 ns;

    j    <= '1' after 4.4 ns when -- a4
                                  (i(31) = '0' and
                                   i(29 downto 26) =
"0000" and
                                   i(5 downto 2) =
"0010") or
                                  -- a23
                                  (i(31) = '0' and
                                   i(29 downto 27) =
"001") else
            '0' after 4.4 ns;

    ill  <= '1' after 4.4 ns when -- a1
                                  (i(31) = '0' and
                                   i(29 downto 26) =
"0000" and
                                   i(5 downto 3) = "000"
and
                                   i(1 downto 0) = "01")
or
                                  -- a3
                                  (i(31) = '0' and
                                   i(29 downto 26) =
"0000" and
                                   i(5 downto 3) = "001"
and
                                   i(1) = '1') or
                                  -- a5
                                  (i(31) = '0' and
                                   i(29 downto 26) =
"0000" and
                                   i(5) = '1' and
                                   i(3) = '1' and
                                   i(1) = '0') or
                                  -- a6
                                  (i(31) = '0' and
                                   i(29 downto 26) =
"0000" and
                                   i(5) = '1' and
                                   i(3 downto 2) = "11")
or
                                  -- a10
                                  (i(31) = '1' and
                                   i(28 downto 26) =
"111") or
                                  -- a11
                                  (i(31) = '0' and
                                   i(29 downto 26) =
"0000" and
                                   i(4) = '1' and
                                   i(2) = '1') or
                                  -- a12
                                  (i(31) = '0' and
                                   i(29 downto 26) =
"0000" and
                                   i(5 downto 4) = "11")
or
                                  -- a16
                                  (i(31) = '1' and
                                   i(29 downto 27) =
"110") or
                                  -- a17
                                  (i(31) = '0' and
                                   i(29 downto 26) =
"0001" and
                                   i(17) = '1') or
                                  -- a18
                                  (i(31) = '0' and
                                   i(29 downto 26) =
"0001" and
                                   i(18) = '1') or
```

```
                                -- a19
                                (i(31) = '0' and
                                 i(29 downto 26) =
"0001" and
                                 i(19) = '1') or
                                -- a28
                                 i(30) = '1' else
            '0' after 4.4 ns;

    ts1  <= '1' after 4.4 ns when -- a4
                                (i(31) = '0' and
                                 i(29 downto 26) =
"0000" and
                                 i(5 downto 2) =
"0010") or
                                -- a15
                                (i(31) = '0' and
                                 i(29 downto 26) =
"0000" and
                                 i(4 downto 3) = "11")
or
                                -- a20
                                (i(27) = '0' and
                                 i(4 downto 3) = "10"
and
                                 i(0) = '1') or
                                -- a22
                                (i(27) = '0' and
                                 i(3 downto 2) = "01")
or
                                -- a25
                                (i(31) = '0' and
                                 i(29 downto 26) =
"0000" and
                                 i(5) = '1') or
                                -- a26
                                (i(31) = '1' and
                                 i(29) = '1') or
                                -- a27
                                (i(31) = '0' and
                                 i(29) = '0' and
                                 i(27 downto 26) =
"01") or
                                -- a29
                                (i(31) = '0' and
                                 i(29 downto 28) =
"01") or
                                -- a30
                                (i(31) = '1' and
                                 i(29) = '0') or
                                -- a31
                                (i(31) = '0' and
                                 i(29) = '1') else
            '0' after 4.4 ns;

    ts2  <= '1' after 4.4 ns when -- a14
                                (i(31) = '0' and
                                 i(29 downto 26) =
"0000" and
                                 i(4 downto 3) = "00")
or
                                -- a15
                                (i(31) = '0' and
                                 i(29 downto 26) =
"0000" and
                                 i(4 downto 3) = "11")
or
                                -- a24
                                (i(31) = '0' and
                                 i(29 downto 27) =
"010") or
                                -- a25
                                (i(31) = '0' and
                                 i(29 downto 26) =
"0000" and
                                 i(5) = '1') or
                                -- a26
                                (i(31) = '1' and
                                 i(29) = '1') else
            '0' after 4.4 ns;

    ttar <= '1' after 4.4 ns when -- a2
                                (i(31) = '0' and
                                 i(29 downto 26) =
"0000" and
                                 i(5 downto 2) =
"0010" and
                                 i(0) = '1') or
                                -- a13
                                (i(31) = '0' and
                                 i(28 downto 26) =
"000" and
                                 i(3) = '0' and
                                 i(0) = '0') or
                                -- a14
                                (i(31) = '0' and
                                 i(29 downto 26) =
"0000" and
                                 i(4 downto 3) = "00")
or
                                -- a21
                                (i(31) = '0' and
                                 i(29 downto 26) =
"0011") or
                                -- a25
                                (i(31) = '0' and
                                 i(29 downto 26) =
"0000" and
                                 i(5) = '1') or
                                -- a30
                                (i(31) = '1' and
                                 i(29) = '0') or
                                -- a31
                                (i(31) = '0' and
                                 i(29) = '1') else
            '0' after 4.4 ns;

    thi  <= '1' after 4.4 ns when -- a8
                                (i(31) = '0' and
                                 i(29 downto 26) =
"0000" and
                                 i(4) = '1' and
                                 i(1) = '0') or
                                -- a15
                                (i(31) = '0' and
                                 i(29 downto 26) =
"0000" and
                                 i(4 downto 3) = "11")
else
            '0' after 4.4 ns;

    tlo  <= '1' after 4.4 ns when -- a7
                                (i(31) = '0' and
                                 i(29 downto 26) =
"0000" and
                                 i(4) = '1' and
                                 i(1) = '1') or
                                -- a15
                                (i(31) = '0' and
                                 i(29 downto 26) =
"0000" and
                                 i(4 downto 3) = "11")
else
            '0' after 4.4 ns;

    exc  <= '1' after 4.4 ns when -- a9
                                (i(31) = '0' and
                                 i(29 downto 26) =
"0000" and
                                 i(3 downto 2) = "11")
else
            '0' after 4.4 ns;

    r    <= '1' after 4.4 ns when -- a4
                                (i(31) = '0' and
                                 i(29 downto 26) =
"0000" and
                                 i(5 downto 2) =
"0010") else
            '0' after 4.4 ns;

    l    <= '1' after 4.4 ns when -- a2
                                (i(31) = '0' and
                                 i(29 downto 26) =
"0000" and
                                 i(5 downto 2) =
"0010" and
                                 i(0) = '1') or
                                -- a21
                                (i(31) = '0' and
                                 i(29 downto 26) =
"0011") else
            '0' after 4.4 ns;

    trs0 <= '1' after 4.4 ns when -- a23
                                (i(31) = '0' and
                                 i(29 downto 27) =
"001") or
                                -- a26
                                (i(31) = '1' and
                                 i(29) = '1') or
                                -- a27
                                (i(31) = '0' and
                                 i(29) = '0' and
                                 i(27 downto 26) =
"01") or
```

```
                                -- a29
                                (i(31) = '0' and
                                 i(29 downto 28) =
"01") else
            '0' after 4.4 ns;

    trs1 <= '1' after 4.4 ns when -- a21
                                (i(31) = '0' and
                                 i(29 downto 26) =
"0011") or
                                -- a30
                                (i(31) = '1' and
                                 i(29) = '0') or
                                -- a31
                                (i(31) = '0' and
                                 i(29) = '1') or
                                -- a32
                                (i(31) = '0' and
                                 i(29 downto 26) =
"0001" and
                                 i(20) = '1') else
            '0' after 4.4 ns;

    add8 <= '1' after 4.4 ns when -- a32
                                i(31) = '0' and
                                i(29 downto 26) =
"0001" and
                                i(20) = '1' else
            '0' after 4.4 ns;

end dfinstdec_a;
```

DFMD

```
library my_packages;
use my_packages.package_1.all;

entity dfmd is
    port(start: in  bit;
         x:     in  bit_32;
         y:     in  bit_32;
         sign:  in  bit;
         divd:  in  bit;
         hi:    out bit_32;
         lo:    out bit_32;
         done1: out bit;
         done2: out bit);
end dfmd;


architecture dfmd_a of dfmd is

    signal temp: bit_64;
    signal dummy: bit;
    signal done_temp2: bit;

begin
    temp <= x * y       when        -- unsigned mult
                        divd = '0' and sign = '0' and
                        dummy = '1' else
            mult(x, y)  when        -- signed mult
                        divd = '0' and sign = '1' and
                        dummy = '1' else
            x / y       when        -- unsigned div
                        divd = '1' and sign = '0' and
                        dummy = '1' else
            div(x, y)   when        -- signed div
                        divd = '1' and sign = '1' and
                        dummy = '1' else
            temp;

    hi <= temp(63 downto 32) after 1000 ns;
    lo <= temp(31 downto 0)  after 1000 ns;

    dummy <= '1' after 1 ns when start = '1' else
             '0' when done_temp2 = '1' else
             dummy;

    done_temp2 <= '1' after 1005 ns when dummy = '1'
else
                  '0' after 1 ns when start'event and
start = '1' else
                  done_temp2;

    done2 <= done_temp2;

    done1 <= '1' after 11 ns when start = '1' else
             '0' after 1 ns ;

end dfmd_a;
```

DFMEM

```
library my_packages;
use my_packages.package_1.all;
use my_packages.dfpack.all;

library df_comp;
use df_comp.dfreg32.all;
use df_comp.dfrslat.all;
use df_comp.dfred.all;
use df_comp.dffed.all;
use df_comp.dfmemdec.all;
use df_comp.dfmu.all;
use df_comp.dfsu.all;
use df_comp.df4to1mux32.all;
use df_comp.dftsb32.all;
use df_comp.df2to1mux3.all;

entity dfmem is
    port (mem_start: in    bit;
          ma:        in    bit;
          inst_in:   in    bit_32;
          data1_in:  in    bit_32;
          data2_in:  in    bit_32;
          addr_bus:  inout bus_bit_32;
          data_bus:  inout bus_bit_32;
          mem_exc:   out   bit;
          mr:        out   bit;
          ml:        out   bit;
          w:         out   bit;
          opcode:    out   bit_3;
          inst_out:  out   bit_32;
          data_out:  out   bit_32;
          vbt:       out   bit_4;
          mem_done:  out   bit);
end dfmem;


architecture dfmem_a of dfmem is

    component dfreg32
        port (d: in  bit_32;
              c: in  bit;
              q: out bit_32);
    end component;

    component dfrslat
        port(s: in  bit;
             r: in  bit;
             q: out bit);
    end component;

    component dfred
        port(i: in  bit;
             o: out bit);
    end component;

    component dffed
        port(i: in  bit;
             o: out bit);
    end component;

    component dfmemdec
        port(start: in  bit;
             inst:  in  bit_32;
             addr:  in  bit_2;
             exc:   out bit;
             vbt:   out bit_4;
             store: out bit;
             load:  out bit;
             sus0:  out bit_2;
             sus1:  out bit_2;
             sus2:  out bit_2;
             sus3:  out bit_2;
             mus0:  out bit;
             mus1:  out bit;
             mus2:  out bit;
             mus3:  out bit;
             done:  out bit);
    end component;

    component dfmu
        port(start:    in  bit;
             inst:     in  bit_32;
             data:     in  bit_32;
             addr:     in  bit_2;
             mus0:     in  bit;
             mus1:     in  bit;
             mus2:     in  bit;
             mus3:     in  bit;
             data_out: out bit_32;
```

```
            done:     out bit);
    end component;

    component dfsu
        port(start:    in  bit;
             mu_data:  in  bit_32;
             sus0:     in  bit_2;
             sus1:     in  bit_2;
             sus2:     in  bit_2;
             sus3:     in  bit_2;
             data_out: out bit_32;
             done:     out bit);
    end component;

    component df4to1mux32
        port(i0: in  bit_32;
             i1: in  bit_32;
             i2: in  bit_32;
             i3: in  bit_32;
             s0: in  bit;
             s1: in  bit;
             o:  out bit_32);
    end component;

    component dftsb32
        port(i: in  bit_32;
             e: in  bit;
             o: out bit_32);
    end component;

    component df2to1mux3
        port (i0: in  bit_3;
              i1: in  bit_3;
              s:  in  bit;
              o:  out bit_3);
    end component;

    signal mem_start_r, mem_start_f: bit;
    signal inst_l, data1_l, data2_l: bit_32;
    signal st, ld, pass, stld, stld_r: bit;
    signal dummy_delay, memdec_done, store, load: bit;
    signal sus0, sus1, sus2, sus3: bit_2;
    signal mus0, mus1, mus2, mus3: bit;
    signal mr_start, mr_start_r: bit;
    signal su_start, su_done: bit;
    signal mu_start: bit;
    signal su_data, mu_data: bit_32;
    signal db_tsb_en, ab_tsb_en: bit;
    signal mrq, mrq_f, ma_r, ma_f, lors: bit;
    signal default_rd_type: bit_3 := "011";
    signal done, done_r: bit;
    signal data_bus_l, tsb_in: bit_32;
    signal tsb_in0, tsb_in1: bit;
begin

    inst_latch: dfreg32
        port map(inst_in, mem_start_r, inst_l);

    inst_out <= inst_l;

    data1_latch: dfreg32
        port map(data1_in, mem_start_r, data1_l);

    data2_latch: dfreg32
        port map(data2_in, mem_start_r, data2_l);

    mem_start_red: dfred
        port map(mem_start, mem_start_r);

    mem_start_fed: dffed
        port map(mem_start, mem_start_f);

    done_latch: dfrslat
        port map(stld_r, mem_start_f, done);

    mem_done <= done;

    mem_done_red: dfred
        port map(done, done_r);

    dummy_delay <= '1' after 2 ns when mem_start = '1'
else
                   '0' after 1 ns;

    memory_decoder: dfmemdec
        port map(dummy_delay, inst_l, data1_l(1 downto
0), mem_exc,
                 vbt, store, load, sus0, sus1, sus2,
sus3,
                 mus0, mus1, mus2, mus3, memdec_done);

    mr_start <= memdec_done and (store or load) after
2.3 ns;

    mr_start_red: dfred
        port map(mr_start, mr_start_r);

    stld <= st or ld or pass after 1.4 ns;

    stld_red: dfred
        port map(stld, stld_r);

    st <= memdec_done and store and mu_start after 1.1
ns;

    ld <= memdec_done and load and su_done after 1.1
ns;

    pass <= memdec_done and (not load) and (not store)
after 1.4 ns;

    mu_start_latch: dfrslat
        port map(mrq_f, done_r, mu_start);

    mu_data_latch: dfreg32
        port map(data_bus, ma_f, data_bus_l);

    mask_unit: dfmu
        port map(mu_start, inst_l, data_bus_l,
data1_l(1 downto 0),
                 mus0, mus1, mus2, mus3, mu_data,
su_start);

    shift_unit: dfsu
        port map(su_start, mu_data, sus0, sus1,
                 sus2, sus3, su_data, su_done);

    data_mux: df4to1mux32
        port map(data1_l, su_data, data2_l, data2_l,
                 load, store, data_out);

    data_bus_tsb: dftsb32
        port map(data2_l, db_tsb_en, data_bus);

    addr_bus_tsb: dftsb32
        port map(tsb_in, ab_tsb_en, addr_bus);

    tsb_in <= data1_l(31 downto 2) & tsb_in1 & tsb_in0;

    tsb_in0 <= data1_l(0) and store;

    tsb_in1 <= data1_l(1) and store;

    db_tsb_en <= store and ma after 1 ns;

    ab_tsb_en <= lors and ma after 1 ns;

    lors <= load or store after 1.3 ns;

    ma_red: dfred
        port map(ma, ma_r);

    ma_fed: dffed
        port map(ma, ma_f);

    mr_latch: dfrslat
        port map(mr_start_r, ma_f, mrq);

    mr <= mrq;

    mrq_fed: dffed
        port map(mrq, mrq_f);

    ml_latch: dfrslat
        port map(ma_r, ma_f, ml);

    w <= store and ma;

    type_mux: df2to1mux3
        port map(default_rd_type, inst_l(28 downto 26),
                 db_tsb_en, opcode);

end dfmem_a;
```

DFMEMDEC

```
library my_packages;
use my_packages.package_1.all;

entity dfmemdec is
    port(start: in  bit;
         inst:  in  bit_32;
         addr:  in  bit_2;
         exc:   out bit;
```

```
        vbt:   out bit_4;
        store: out bit;
        load:  out bit;
        sus0:  out bit_2;
        sus1:  out bit_2;
        sus2:  out bit_2;
        sus3:  out bit_2;
        mus0:  out bit;
        mus1:  out bit;
        mus2:  out bit;
        mus3:  out bit;
        done:  out bit);
end dfmemdec;

architecture dfmemdec_a of dfmemdec is

    signal vbt0, vbt1, vbt2, vbt3: bit;
    signal sus00, sus01, sus10, sus11: bit;
    signal sus20, sus21, sus30, sus31: bit;

begin

    done <= '1' after 5.3 ns when start = '1' else
            '0' after 1 ns;

    exc <= '1'    after 4.3 ns when -- a3
                                    (inst(31) = '1' and
                                     inst(29) = '0' and
                                     inst(27 downto 26) =
"11" and
                                     addr(1) = '1') or
                                    -- a8
                                    (inst(31) = '1' and
                                     inst(27 downto 26) =
"01" and
                                     addr(0) = '1') or
                                    -- a10
                                    (inst(31) = '1' and
                                     inst(29) = '0' and
                                     inst(26) = '1' and
                                     addr(0) = '1')
                                else
           '0'    after 4.3 ns;

    vbt0 <= '1' after 4.3 ns when -- a9
                                    (inst(31) = '1' and
                                     inst(29 downto 27) =
"011" and
                                     addr = "00") or
                                    -- a13
                                    (inst(31) = '1' and
                                     inst(29 downto 27) =
"011" and
                                     addr = "01") or
                                    -- a23
                                    (inst(28 downto 27) =
"01" and
                                     addr = "00") or
                                    -- a19
                                    (inst(31) = '1' and
                                     inst(29 downto 27) =
"011" and
                                     addr = "10") or
                                    -- a17
                                    (inst(31) = '1' and
                                     inst(29) = '0' and
                                     inst(27) = '0' and
                                     addr = "10") or
                                    -- a18
                                    (inst(31) = '1' and
                                     inst(29) = '0' and
                                     inst(27) = '0' and
                                     addr = "01") or
                                    -- a21
                                    (inst(31) = '1' and
                                     inst(29) = '0' and
                                     inst(27) = '0' and
                                     addr = "11") or
                                    -- a22
                                    (inst(31) = '1' and
                                     inst(29) = '0' and
                                     inst(27) = '0' and
                                     addr = "00") or
                                    -- a24
                                    (inst(31) = '1' and
                                     inst(29 downto 27) =
"011" and
                                     addr = "11") or
                                    -- a25
                                    (inst(31) = '0')
                                else
           '0' after 4.3 ns;

    vbt1 <= '1' after 4.3 ns when   -- a13
                                    (inst(31) = '1' and
                                     inst(29 downto 27) =
"011" and
                                     addr = "01") or
                                    -- a23
                                    (inst(28 downto 27) =
"01" and
                                     addr = "00") or
                                    -- a15
                                    (inst(31) = '1' and
                                     inst(28 downto 27) =
"01" and
                                     addr = "01") or
                                    -- a19
                                    (inst(31) = '1' and
                                     inst(29 downto 27) =
"011" and
                                     addr = "10") or
                                    -- a17
                                    (inst(31) = '1' and
                                     inst(29) = '0' and
                                     inst(27) = '0' and
                                     addr = "10") or
                                    -- a18
                                    (inst(31) = '1' and
                                     inst(29) = '0' and
                                     inst(27) = '0' and
                                     addr = "01") or
                                    -- a21
                                    (inst(31) = '1' and
                                     inst(29) = '0' and
                                     inst(27) = '0' and
                                     addr = "11") or
                                    -- a22
                                    (inst(31) = '1' and
                                     inst(29) = '0' and
                                     inst(27) = '0' and
                                     addr = "00") or
                                    -- a24
                                    (inst(31) = '1' and
                                     inst(29 downto 27) =
"011" and
                                     addr = "11") or
                                    -- a25
                                    (inst(31) = '0')
                                else
           '0' after 4.3 ns;

    vbt2 <= '1' after 4.3 ns when   -- a11
                                    (inst(28 downto 27) =
"01" and
                                     addr(0) = '0') or
                                    -- a15
                                    (inst(31) = '1' and
                                     inst(28 downto 27) =
"01" and
                                     addr = "01") or
                                    -- a19
                                    (inst(31) = '1' and
                                     inst(29 downto 27) =
"011" and
                                     addr = "10") or
                                    -- a17
                                    (inst(31) = '1' and
                                     inst(29) = '0' and
                                     inst(27) = '0' and
                                     addr = "10") or
                                    -- a18
                                    (inst(31) = '1' and
                                     inst(29) = '0' and
                                     inst(27) = '0' and
                                     addr = "01") or
                                    -- a21
                                    (inst(31) = '1' and
                                     inst(29) = '0' and
                                     inst(27) = '0' and
                                     addr = "11") or
                                    -- a22
                                    (inst(31) = '1' and
                                     inst(29) = '0' and
                                     inst(27) = '0' and
                                     addr = "00") or
                                    -- a24
                                    (inst(31) = '1' and
                                     inst(29 downto 27) =
 "011" and
                                     addr = "11") or
                                    -- a25
                                    (inst(31) = '0')
                                else
           '0' after 4.3 ns;
```

```
    vbt3 <= '1' after 4.3 ns   when -- a12
                                    (inst(31) = '1' and
                                     inst(29 downto 27) =
"001") or
                                    -- a17
                                    (inst(31) = '1' and
                                     inst(29) = '0' and
                                     inst(27) = '0' and
                                     addr = "10") or
                                    -- a18
                                    (inst(31) = '1' and
                                     inst(29) = '0' and
                                     inst(27) = '0' and
                                     addr = "01") or
                                    -- a21
                                    (inst(31) = '1' and
                                     inst(29) = '0' and
                                     inst(27) = '0' and
                                     addr = "11") or
                                    -- a22
                                    (inst(31) = '1' and
                                     inst(29) = '0' and
                                     inst(27) = '0' and
                                     addr = "00") or
                                    -- a24
                                    (inst(31) = '1' and
                                     inst(29 downto 27) =
"011" and
                                     addr = "11") or
                                    -- a25
                                    (inst(31) = '0')
                               else
          '0' after 4.3 ns;

    store <= '1'  after 4.3 ns when -- a14
                                    (inst(31) = '1' and
                                     inst(29) = '1')
                               else
            '0'  after 4.3 ns;

    load <= '1'   after 4.3 ns when -- a9
                                    (inst(31) = '1' and
                                     inst(29 downto 27) =
"011" and
                                     addr = "00") or
                                    -- a12
                                    (inst(31) = '1' and
                                     inst(29 downto 27) =
"001") or
                                    -- a13
                                    (inst(31) = '1' and
                                     inst(29 downto 27) =
"011" and
                                     addr = "01") or
                                    -- a17
                                    (inst(31) = '1' and
                                     inst(29) = '0' and
                                     inst(27) = '0' and
                                     addr = "10") or
                                    -- a18
                                    (inst(31) = '1' and
                                     inst(29) = '0' and
                                     inst(27) = '0' and
                                     addr = "01") or
                                    -- a19
                                    (inst(31) = '1' and
                                     inst(29 downto 27) =
"011" and
                                     addr = "10") or
                                    -- a21
                                    (inst(31) = '1' and
                                     inst(29) = '0' and
                                     inst(27) = '0' and
                                     addr = "11") or
                                    -- a22
                                    (inst(31) = '1' and
                                     inst(29) = '0' and
                                     inst(27) = '0' and
                                     addr = "00") or
                                    -- a24
                                    (inst(31) = '1' and
                                     inst(29 downto 27) =
"011" and
                                     addr = "11")
                               else
            '0'   after 4.3 ns;

    sus00 <= '1'  after 4.3 ns when -- a12
                                    (inst(31) = '1' and
                                     inst(29) = '0' and
                                     inst(27 downto 26) =
"00" and
                                     addr = "10") or
                                    -- a19
                                    (inst(31) = '1' and
                                     inst(29 downto 27) =
"011" and
                                     addr = "10") or
                                    -- a9
                                    (inst(31) = '1' and
                                     inst(29 downto 27) =
"011" and
                                     addr = "00") or
                                    -- a17
                                    (inst(31) = '1' and
                                     inst(29) = '0' and
                                     inst(27 downto 26) =
"00" and
                                     addr = "00")
                               else
          '0'  after 4.3 ns;

    sus01 <= '1'  after 4.3 ns when -- a13
                                    (inst(31) = '1' and
                                     inst(29 downto 27) =
"011" and
                                     addr = "01") or
                                    -- a17
                                    (inst(31) = '1' and
                                     inst(29) = '0' and
                                     inst(27 downto 26) =
"00" and
                                     addr = "00") or
                                    -- a9
                                     (inst(31) = '1' and
                                     inst(29 downto 27) =
"011" and
                                     addr = "00") or
                                    -- a22
                                    (inst(31) = '1' and
                                     inst(29) = '0' and
                                     inst(27 downto 26) =
"01" and
                                     addr(1) = '0') or
                                    -- a18
                                    (inst(31) = '1' and
                                     inst(29) = '0' and
                                     inst(27) = '0' and
                                     addr = "01")
                               else
          '0'  after 4.3 ns;

    sus10 <= '1'  after 4.3 ns when -- a22
                                    (inst(31) = '1' and
                                     inst(29) = '0' and
                                     inst(27 downto 26) =
"01" and
                                     addr(1) = '0') or
                                    -- a23
                                    (inst(31) = '1' and
                                     inst(29) = '0' and
                                     inst(26) = '1' and
                                     addr(1) = '1') or
                                    -- a24
                                    (inst(31) = '1' and
                                     inst(29 downto 27) =
"011" and
                                     addr = "11") or
                                    -- a25
                                    (inst(31) = '0') or
                                    -- a13
                                    (inst(31) = '1' and
                                     inst(29 downto 27) =
"011" and
                                     addr = "01") or
                                    -- a16
                                    (inst(28 downto 27) =
"01" and
                                     addr = "00") or
                                    -- a17
                                    (inst(31) = '1' and
                                     inst(29) = '0' and
                                     inst(27 downto 26) =
"00" and
                                     addr = "00") or
                                    -- a21
                                    (inst(31) = '1' and
                                     inst(29) = '0' and
                                     inst(27) = '0' and
                                     addr = "11")
                               else
          '0'  after 4.3 ns;
```

```
    sus11 <= '1'  after 4.3 ns when -- a19
                                (inst(31) = '1' and
                                 inst(29 downto 27) =
"011" and
                                 addr = "10") or
                                -- a13
                                (inst(31) = '1' and
                                 inst(29 downto 27) =
"011" and
                                 addr = "01") or
                                -- a22
                                (inst(31) = '1' and
                                 inst(29) = '0' and
                                 inst(27 downto 26) =
"01" and
                                 addr(1) = '0')
                            else
            '0'  after 4.3 ns;

    sus20 <= '1'  after 4.3 ns when -- a11
                                (inst(31) = '1' and
                                 inst(28 downto 27) =
"01" and
                                 addr = "01") or
                                -- a17
                                (inst(31) = '1' and
                                 inst(29) = '0' and
                                 inst(27 downto 26) =
"00" and
                                 addr = "00") or
                                -- a19
                                (inst(31) = '1' and
                                 inst(29 downto 27) =
"011" and
                                 addr = "10") or
                                -- a23
                                (inst(31) = '1' and
                                 inst(29) = '0' and
                                 inst(26) = '1' and
                                 addr(1) = '1') or
                                -- a21
                                (inst(31) = '1' and
                                 inst(29) = '0' and
                                 inst(27) = '0' and
                                 addr = "11")
                            else
            '0'  after 4.3 ns;

    sus21 <= '1'  after 4.3 ns when -- a19
                                (inst(31) = '1' and
                                 inst(29 downto 27) =
"011" and
                                 addr = "10") or
                                -- a16
                                (inst(28 downto 27) =
"01" and
                                 addr = "00") or
                                -- a24
                                (inst(31) = '1' and
                                 inst(29 downto 27) =
"011" and
                                 addr = "11") or
                                -- a25
                                (inst(31) = '0') or
                                -- a23
                                (inst(31) = '1' and
                                 inst(29) = '0' and
                                 inst(26) = '1' and
                                 addr(1) = '1')
                            else
            '0'  after 4.3 ns;

    sus30 <= '1'  after 4.3 ns when -- a20
                                (inst(28 downto 27) =
"01" and
                                 addr(0) = '0') or
                                -- a21
                                (inst(31) = '1' and
                                 inst(29) = '0' and
                                 inst(27) = '0' and
                                 addr = "11") or
                                -- a23
                                (inst(31) = '1' and
                                 inst(29) = '0' and
                                 inst(26) = '1' and
                                 addr(1) = '1') or
                                -- a17-
                                (inst(31) = '1' and
                                 inst(29) = '0' and
                                 inst(27 downto 26) =
"00" and
                                 addr = "00") or
                                -- a24-
                                (inst(31) = '1' and
                                 inst(29 downto 27) =
"011" and
                                 addr = "11") or
                                -- a25-
                                (inst(31) = '0')
                            else
            '0'  after 4.3 ns;

    sus31 <= '1'  after 4.3 ns when -- a16
                                (inst(28 downto 27) =
"01" and
                                 addr = "00") or
                                -- a11
                                (inst(31) = '1' and
                                 inst(28 downto 27) =
"01" and
                                 addr = "01") or
                                -- a23
                                (inst(31) = '1' and
                                 inst(29) = '0' and
                                 inst(26) = '1' and
                                 addr(1) = '1') or
                                -- a24
                                (inst(31) = '1' and
                                 inst(29 downto 27) =
"011" and
                                 addr = "11") or
                                -- a25
                                (inst(31) = '0')
                            else
            '0'  after 4.3 ns;

    mus3 <= '1'   after 4.3 ns when -- a22
                                (inst(31) = '1' and
                                 inst(29) = '0' and
                                 inst(27) = '0' and
                                 addr = "00") or
                                -- a24
                                (inst(31) = '1' and
                                 inst(29 downto 27) =
"011" and
                                 addr = "11") or
                                -- a25
                                (inst(31) = '0') or
                                -- a17
                                (inst(31) = '1' and
                                 inst(29) = '0' and
                                 inst(27 downto 26) =
"00" and
                                 addr = "00") or
                                -- a19
                                (inst(31) = '1' and
                                 inst(29 downto 27) =
"011" and
                                 addr = "10") or
                                -- a13
                                (inst(31) = '1' and
                                 inst(29 downto 27) =
"011" and
                                 addr = "01") or
                                -- a16
                                (inst(28 downto 27) =
"01" and
                                 addr = "00") or
                                -- a9
                                 (inst(31) = '1' and
                                 inst(29 downto 27) =
"011" and
                                 addr = "00")
                            else
            '0'   after 4.3 ns;

    mus2 <= '1'   after 4.3 ns when -- a11
                                (inst(31) = '1' and
                                 inst(28 downto 27) =
"01" and
                                 addr = "01") or
                                -- a13
                                (inst(31) = '1' and
                                 inst(29 downto 27) =
"011" and
                                 addr = "01") or
                                -- a18
                                (inst(31) = '1' and
                                 inst(29) = '0' and
                                 inst(27) = '0' and
                                 addr = "01") or
                                -- a19
                                (inst(31) = '1' and
                                 inst(29 downto 27) =
"011" and
                                 addr = "10") or
```

```
                                -- a22
                                (inst(31) = '1' and
                                 inst(29) = '0' and
                                 inst(27) = '0' and
                                 addr = "00") or
                                -- a24
                                (inst(31) = '1' and
                                 inst(29 downto 27) =
"011" and
                                 addr = "11") or
                                -- a25
                                (inst(31) = '0') or
                                -- a16
                                (inst(28 downto 27) =
"01" and
                                 addr = "00")
                            else
            '0'   after 4.3 ns;

    mus1 <= '1'   after 4.3 ns when -- a11
                                (inst(31) = '1' and
                                 inst(28 downto 27) =
"01" and
                                 addr = "01") or
                                -- a12
                                (inst(31) = '1' and
                                 inst(29) = '0' and
                                 inst(27 downto 26) =
"00" and
                                 addr = "10") or
                                -- a20
                                (inst(28 downto 27) =
"01" and
                                 addr(0) = '0') or
                                -- a19
                                (inst(31) = '1' and
                                 inst(29 downto 27) =
"011" and
                                 addr = "10") or
                                -- a23
                                (inst(31) = '1' and
                                 inst(29) = '0' and
                                 inst(26) = '1' and
                                 addr(1) = '1') or
                                -- a24
                                (inst(31) = '1' and
                                 inst(29 downto 27) =
"011" and
                                 addr = "11") or
                                -- a25
                                (inst(31) = '0')
                            else
            '0'   after 4.3 ns;

    mus0 <= '1'   after 4.3 ns when -- a15
                                (inst(31) = '1' and
                                 inst(29 downto 27) =
"001") or
                                -- a21
                                (inst(31) = '1' and
                                 inst(29) = '0' and
                                 inst(27) = '0' and
                                 addr = "11") or
                                -- a23
                                (inst(31) = '1' and
                                 inst(29) = '0' and
                                 inst(26) = '1' and
                                 addr(1) = '1') or
                                -- a24
                                (inst(31) = '1' and
                                 inst(29 downto 27) =
"011" and
                                 addr = "11") or
                                -- a25
                                (inst(31) = '0')
                            else
            '0'   after 4.3 ns;

    vbt <= vbt3 & vbt2 & vbt1 & vbt0;

    sus0 <= sus01 & sus00;

    sus1 <= sus11 & sus10;

    sus2 <= sus21 & sus20;

    sus3 <= sus31 & sus30;

end dfmemdec_a;
```

DFMEMORY

```
use std.textio.all;

library my_packages;
use my_packages.package_1.all;
use my_packages.dfpack.all;

entity dfmemory is
    port(load:     in    question_type;
         req:      in    bit;
         w:        in    bit;
         opcode:   in    bit_3;
         ack:      out   bit;
         load_ack: out   question_type;
         addr_bus: inout bus_bit_32 bus;
         data_bus: inout bus_bit_32 bus);
end dfmemory;


architecture dfmemory_a of dfmemory is

    signal ack_temp: bit;

    signal addr_temp: integer;
    signal m4000, m4004, m4008, m400c, m4010: bit_32;
    signal m4014, m4018, m401c, m4020, m4024: bit_32;

begin

    memory: process
        constant low_address: integer := 0;
        constant high_address: integer := 65535;
        type memory_array is
            array (integer range low_address to
high_address) of bit_32;
        variable mem: memory_array;
        variable addr: integer range 0 to high_address;
        variable temp: bit_32;

        file in1: text is in "machine";
        variable line1: line;
        variable inst: bit_32;

        variable skip: question_type;
        variable addr_word: bit_30;
        variable addr_byte: bit_2;

    begin

        wait on req, load;

        if skip = no then
            if load = yes then
                addr := 0;
                while endfile(in1) = false loop
                    readline(in1, line1);
                    read(line1, inst);
                    mem(addr) := inst;
                    addr := addr + 1;
                end loop;
                skip := yes;
                load_ack <= yes after delay;
            end if;
        else
          if req = '1' then
            data_bus <= null;
            addr_word := addr_bus(31 downto 2);
            addr_byte := addr_bus(1 downto 0);

            -- send acknowledge signal to bus
controller
            ack <= '1' after 5 ns;
            ack_temp <= '1' after 5 ns;

            assert addr_word(29 downto 14) = x"0000"
                report "ADDRESS OUT OF RANGE";

            if addr_word(29 downto 14) = x"0000" then
                if w = '0' then
                    addr := bton(addr_word);
                    temp := mem(addr);

                    -- read operations
                    if opcode = m_lb then
                        case addr_byte is
                          when b"00" =>
                            data_bus <= se8to32(temp(31
downto 24));
                          when b"01" =>
                            data_bus <= se8to32(temp(23
downto 16));
                          when b"10" =>
```

```
                        data_bus <= se8to32(temp(15
downto 8));
                      when b"11" =>
                        data_bus <= se8to32(temp(7
downto 0));
                    end case;

                  elsif opcode = m_lh then
                    case addr_byte is
                      when b"00" =>
                        data_bus <=
se16to32(temp(31 downto 16));
                      when b"10" =>
                        data_bus <=
se16to32(temp(15 downto 0));
                      when others =>
                        -- flag address exception
                        null;
                    end case;

                  elsif opcode = m_lwl then
                    case addr_byte is
                      when b"00" =>
                        data_bus <= temp;
                      when b"01" =>
                        data_bus(23 downto 0) <=
temp(23 downto 0);
                        data_bus(31 downto 24) <=
x"00";
                      when b"10" =>
                        data_bus(15 downto 0) <=
temp(15 downto 0);
                        data_bus(31 downto 16) <=
x"0000";
                      when b"11" =>
                        data_bus(7 downto 0) <=
temp(7 downto 0);
                        data_bus(31 downto 8) <=
x"000000";
                    end case;

                  elsif opcode = m_lw then
                    case addr_byte is
                      when b"00" =>
                        data_bus <= temp;

                      when others =>
                        -- flag address exception
                        null;
                    end case;

                  elsif opcode = m_lbu then
                    case addr_byte is
                      when b"00" =>
                        data_bus <= x"000000" &
temp(31 downto 24);
                      when b"01" =>
                        data_bus <= x"000000" &
temp(23 downto 16);
                      when b"10" =>
                        data_bus <= x"000000" &
temp(15 downto 8);
                      when b"11" =>
                        data_bus <= x"000000" &
temp(7 downto 0);
                    end case;

                  elsif opcode = m_lhu then
                    case addr_byte is
                      when b"00" =>
                        data_bus <= x"0000" &
temp(31 downto 16);
                      when b"10" =>
                        data_bus <= x"0000" &
temp(15 downto 0);
                      when others =>
                        -- flag address exception
                        null;
                    end case;

                  elsif opcode = m_lwr then
                    case addr_byte is
                      when b"00" =>
                        data_bus(31 downto 24) <=
temp(31 downto 24);
                        data_bus(23 downto 0) <=
x"000000";
                      when b"01" =>
                        data_bus(31 downto 16) <=
temp(31 downto 16);
                        data_bus(15 downto 0) <=
x"0000";
                      when b"10" =>
                        data_bus(31 downto 8) <=
temp(31 downto 8);
                        data_bus(7 downto 0) <=
x"00";
                      when b"11" =>
                        data_bus <= temp;
                    end case;

                  else
                    -- reserved instructions
                    null;

                  end if;   -- end read operations

                else
                  addr := bton(addr_word);

                  -- write operations
                  if opcode = m_sb then
                    case addr_byte is
                      when b"00" =>
                        mem(addr)(31 downto 24) :=
data_bus(7 downto 0);
                      when b"01" =>
                        mem(addr)(23 downto 16) :=
data_bus(7 downto 0);
                      when b"10" =>
                        mem(addr)(15 downto 8) :=
data_bus(7 downto 0);
                      when b"11" =>
                        mem(addr)(7 downto 0) :=
data_bus(7 downto 0);
                    end case;

                  elsif opcode = m_sh then
                    case addr_byte is
                      when b"00" =>
                        mem(addr)(31 downto 16) :=
data_bus(15 downto 0);
                      when b"10" =>
                        mem(addr)(15 downto 0) :=
data_bus(15 downto 0);
                      when others =>
                        -- flag address exception
                        null;
                    end case;

                  elsif opcode = m_swl then
                    case addr_byte is
                      when b"00" =>
                        mem(addr) := data_bus;
                      when b"01" =>
                        mem(addr)(23 downto 0) :=
data_bus(31 downto 8);
                      when b"10" =>
                        mem(addr)(15 downto 0) :=
data_bus(31 downto 16);
                      when b"11" =>
                        mem(addr)(7 downto 0) :=
data_bus(31 downto 24);
                    end case;

                  elsif opcode = m_sw then
                    case addr_byte is
                      when b"00" =>
                        mem(addr) := data_bus;
                      when others =>
                        -- flag address exception
                        null;
                    end case;

                  elsif opcode = m_swr then
                    case addr_byte is
                      when b"00" =>
                        mem(addr)(31 downto 24) :=
data_bus(7 downto 0);
                      when b"01" =>
                        mem(addr)(31 downto 16) :=
data_bus(15 downto 0);
                      when b"10" =>
                        mem(addr)(31 downto 8) :=
data_bus(23 downto 0);
                      when b"11" =>
                        mem(addr) := data_bus;
                    end case;

                  else
                    -- reserved instructions
                    null;

                  end if; -- end write operations

                end if;   --end read/write selection
```

```
                addr_temp <= addr;
                m4000 <= mem(4096);
                m4004 <= mem(4097);
                m4008 <= mem(4098);
                m400c <= mem(4099);
                m4010 <= mem(4100);
                m4014 <= mem(4101);
                m4018 <= mem(4102);
                m401c <= mem(4103);
                m4020 <= mem(4104);
                m4024 <= mem(4105);

            end if;   -- end if address is in range

            --wait on ack;
            wait on ack_temp;
            ack <= '0' after 50 ns;

          else -- req = '0'
            data_bus <= null;
            ack_temp <= '0';

          end if;   -- if req = '1'

        end if;    -- end if skip no

    end process memory;

end dfmemory_a;
```

DFMU

```
library my_packages;
use my_packages.package_1.all;

library df_comp;
use df_comp.df2to1mux8.all;

entity dfmu is
    port(start:    in  bit;
         inst:     in  bit_32;
         data:     in  bit_32;
         addr:     in  bit_2;
         mus0:     in  bit;
         mus1:     in  bit;
         mus2:     in  bit;
         mus3:     in  bit;
         data_out: out bit_32;
         done:     out bit);
end dfmu;


architecture dfmu_a of dfmu is

    component df2to1mux8
        port (i0: in  bit_8;
              i1: in  bit_8;
              s:  in  bit;
              o:  out bit_8);
    end component;

    signal se_byte: bit_8;
    signal se_nibble: bit_4;
    signal seb: bit;

begin

    done <= '1' after 3.4 ns when start = '1' else
            '0' after 1 ns;

    mux0: df2to1mux8
        port map(se_byte, data(7 downto 0), mus0,
data_out(7 downto 0));

    mux1: df2to1mux8
        port map(se_byte, data(15 downto 8), mus1,
data_out(15 downto 8));

    mux2: df2to1mux8
        port map(se_byte, data(23 downto 16), mus2,
data_out(23 downto 16));

    mux3: df2to1mux8
        port map(se_byte, data(31 downto 24), mus3,
data_out(31 downto 24));

    se_byte <= se_nibble & se_nibble;

    se_nibble <= seb & seb & seb & seb;

    seb  <= '1'   after 2.1 ns when -- a1
                                    (inst(28) = '0' and
                                     data(15) = '1' and
                                     addr = "10") or
                                    -- a2
                                    (inst(28) = '0' and
                                     data(31) = '1' and
                                     addr = "00") or
                                    -- a3
                                    (inst(28) = '0' and
                                     data(7) = '1' and
                                     addr = "11") or
                                    -- a4
                                    (inst(28) = '0' and
                                     data(23) = '1' and
                                     addr = "01")
                                else
            '0'   after 2.1 ns;

end dfmu_a;
```

DFOUTSEL

```
library my_packages;
use my_packages.package_1.all;

library df_comp;
use df_comp.df4to1mux32.all;

entity dfoutsel is
    port(inst:   in  bit_32;
         alu:    in  bit_32;
         add8:   in  bit_32;
         hilo:   in  bit_32;
         output: out bit_32);
end dfoutsel;


architecture dfoutsel_a of dfoutsel is

    component df4to1mux32
        port(i0: in  bit_32;
             i1: in  bit_32;
             i2: in  bit_32;
             i3: in  bit_32;
             s0: in  bit;
             s1: in  bit;
             o:  out bit_32);
    end component;

    signal s0, s1: bit;
begin

    mux: df4to1mux32
        port map(alu, add8, hilo, hilo, s0, s1,
output);

    s0 <= '1' after 1.6 ns when inst(31 downto 26) =
"000001" else
          '0' after 1.6 ns;

    s1 <= '1' after 2.3 ns when inst(31 downto 26) =
"000000" and
                                inst(5 downto 4) = "01"
and
                                inst(0) = '0' else
          '0' after 2.3 ns;

end dfoutsel_a;
```

DFOVRF

```
library my_packages;
use my_packages.package_1.all;

entity dfovrf is
    port(inst:     in  bit_32;
         data:     in  bit_32;
         carry:    in  bit;
         overflow: out bit);
end dfovrf;


architecture dfovrf_a of dfovrf is
begin
    overflow <= '1' after 4.6 ns
            when
                carry = '1'
            and
```

```
                (inst(31 downto 26) = b"001000"  --
addi instruction
                or
                  (inst(31 downto 26) = b"000000" and
                   inst(5 downto 2) = b"1000" and
                   inst(0) = '0')                  -- add
or sub instruction
                )
            else
                '0' after 4.6 ns;
end dfovrf_a;
```

DFRED

```
entity dfred is
    port(i: in bit;
         o: out bit);
end dfred;


architecture dfred_a of dfred is
    signal temp: bit;
begin
    temp <= '1' after 0.3 ns when i'event and i = '1'
else
            '0' after 1.1 ns when temp = '1' else
            temp;

    o <= temp;

end dfred_a;
```

DFREG

```
entity dfreg is
    port(d: in  bit;
         c: in  bit;
         q: out bit);
end dfreg;


architecture dfreg_a of dfreg is
    signal temp: bit;
begin
    temp <= d after 0.9 ns when c'event and c = '1'
else
            temp;
    q    <= temp;
end dfreg_a;
```

DFREG32

```
library my_packages;
use my_packages.package_1.all;

entity dfreg32 is
    port (d: in  bit_32;
          c: in  bit;
          q: out bit_32);
end dfreg32;


architecture dfreg32_a of dfreg32 is
    signal temp: bit_32;
begin
    temp <= d after 0.9 ns when c'event and c = '1'
else
            temp;
    q    <= temp;
end dfreg32_a;
```

DFREG4

```
library my_packages;
use my_packages.package_1.all;

entity dfreg4 is
    port (d: in  bit_4;
          c: in  bit;
          q: out bit_4);
end dfreg4;


architecture dfreg4_a of dfreg4 is
    signal temp: bit_4;
begin
    temp <= d after 0.9 ns when c'event and c = '1'
else
            temp;
    q    <= temp;
end dfreg4_a;
```

DFREG5

```
library my_packages;
use my_packages.package_1.all;

entity dfreg5 is
    port (d: in  bit_5;
          c: in  bit;
          q: out bit_5);
end dfreg5;


architecture dfreg5_a of dfreg5 is
    signal temp: bit_5;
begin
    temp <= d after 0.9 ns when c'event and c = '1'
else
            temp;
    q    <= temp;
end dfreg5_a;
```

DFREGBANK

```
library my_packages;
use my_packages.package_1.all;

entity dfregbank is
    port(data:     in  bit_32;
         reg_sel:  in  bit_5;
         a_sel:    in  bit_5;
         b_sel:    in  bit_5;
         db_sel:   in  bit_5;
         write:    in  bit;
         start:    in  bit;
         vbt:      in  bit_4;
         a:        out bit_32;
         b:        out bit_32;
         db:       out bit_32;
         r1_test:  out bit_32;
         r2_test:  out bit_32;
         r3_test:  out bit_32;
         r4_test:  out bit_32;
         r31_test: out bit_32);
end dfregbank;


architecture dfregbank_a of dfregbank is
    type register_byte is array (0 to 31) of bit_8;
    signal r0, r1, r2, r3: register_byte;
    signal an, bn, regn, dbn: bit_5_range;
    signal db_id, db_wb: bit_32;
    signal d0, d1, d2, d3: bit_8;
begin
    an   <= bton(a_sel);
    bn   <= bton(b_sel);
    regn <= bton(reg_sel);
    dbn  <= bton(db_sel);


    d0 <= data(7 downto 0);
    d1 <= data(15 downto 8);
    d2 <= data(23 downto 16);
    d3 <= data(31 downto 24);


    r0(regn) <= d0 after 0.3 ns when
                    regn /= 0 and vbt(0) = '1' and
                    write'event and write = '1' else
                r0(regn);

    r1(regn) <= d1 after 0.3 ns when
                    regn /= 0 and vbt(1) = '1' and
                    write'event and write = '1' else
                r1(regn);

    r2(regn) <= d2 after 0.3 ns when
                    regn /= 0 and vbt(2) = '1' and
                    write'event and write = '1' else
                r2(regn);

    r3(regn) <= d3 after 0.3 ns when
                    regn /= 0 and vbt(3) = '1' and
```

```
                write'event and write = '1' else
            r3(regn);

    a <= r3(an) & r2(an) & r1(an) & r0(an) after 0.6
ns;
    b <= r3(bn) & r2(bn) & r1(bn) & r0(bn) after 0.6
ns;

    db_id(dbn) <= db_id(dbn) xor '1' after 1 ns when
                        start'event and start =
'1' else
                db_id(dbn);

    db_wb(regn) <= db_wb(regn) xor '1' after 1 ns when
                        write'event and write =
'1' else
                db_wb(regn);

    db <= (db_id xor db_wb) and x"fffffffe" after 1 ns;

    r1_test  <= r3(1) & r2(1) & r1(1) & r0(1);
    r2_test  <= r3(2) & r2(2) & r1(2) & r0(2);
    r3_test  <= r3(3) & r2(3) & r1(3) & r0(3);
    r4_test  <= r3(4) & r2(4) & r1(4) & r0(4);
    r31_test <= r3(31) & r2(31) & r1(31) & r0(31);

end dfregbank_a;
```

DFREGR

```
entity dfregr is
    port(d: in  bit;
         c: in  bit;
         r: in  bit;
         q: out bit);
end dfregr;


architecture dfregr_a of dfregr is
    signal temp: bit;
begin
    temp <= d after 0.9 ns when c'event and c = '1' and
r = '0' else
            '0' after 0.3 ns when r = '1' else
            temp;
    q    <= temp;
end dfregr_a;
```

DFRSLAT

```
entity dfrslat is
    port(s: in  bit;
         r: in  bit;
         q: out bit);
end dfrslat;


architecture dfrslat_a of dfrslat is
    signal temp: bit;
begin
    temp <= '1' after 1 ns when s = '1' and r = '0'
else
            '0' after 0.4 ns when s = '0' and r = '1'
else
          temp;
    q    <= temp;
end dfrslat_a;
```

DFSCTL

```
library my_packages;
use my_packages.package_1.all;

entity dfsctl is
    port(inst: in  bit_32;
         a:    in  bit_5;
         lr:   out bit;
         la:   out bit;
         sel:  out bit_5);
end dfsctl;


architecture dfsctl_a of dfsctl is
    signal s, lui: bit;
begin

    s   <= '1' when inst(31 downto 26) = "000000" and
                    inst(5 downto 3) = "000" else
           '0';

    lui <= '1' when inst(31 downto 26) = "001111" else
           '0';

    lr <= '0' after 2.7 ns when lui = '1' else
          inst(1) after 1 ns;

    la <= '0' after 2.7 ns when lui = '1' else
          inst(0) after 1 ns;

    sel <= "10000"           after 2.7 ns when
                               lui = '1' else
           inst(10 downto 6) after 2.7 ns when
                               s   = '1' and inst(2) =
'0' else
           a                 after 2.7 ns when
                               s   = '1' and inst(2) =
'1' else
           "00000"           after 2.7 ns;

end dfsctl_a;
```

DFSHIFT

```
library my_packages;
use my_packages.package_1.all;

entity dfshift is
    port(i:   in  bit_32;
         lr:  in  bit;
         la:  in  bit;
         sel: in  bit_5;
         o:   out bit_32);
end dfsbift;


architecture dfshift_a of dfshift is
begin

    o <= shift_ll(i, bton(sel)) after 3.3 ns when
                                        lr = '0' and la
= '0' else
         shift_rl(i, bton(sel)) after 3.3 ns when
                                        lr = '1' and la
= '0' else
         shift_ra(i, bton(sel)) after 3.3 ns when
                                        lr = '1' and la
= '1' else
         i after 3.3 ns;

end dfshift_a;
```

DFSLT

```
library my_packages;
use my_packages.package_1.all;

entity dfslt is
    port(i:    in  bit_32;
         inst: in  bit_32;
         ltz:  in  bit;
         o:    out bit_32);
end dfslt;


architecture dfslt_a of dfslt is
    signal temp: bit_32;
begin

    temp <= x"0000000" & "000" & ltz;

    o <= temp after 3.5 ns when -- slti, sltiu
                                inst(31 downto 27) =
"00101" or
                                -- slt, sltu
                                (inst(31 downto 26) =
"000000" and
                                 inst(5 downto 1) =
"10101") else
         i after 3.5 ns;

end dfslt_a;
```

DFSU

```
library my_packages;
use my_packages.package_1.all;

library df_comp;
use df_comp.df4to1mux8.all;

entity dfsu is
    port(start:    in  bit;
         mu_data:  in  bit_32;
         sus0:     in  bit_2;
         sus1:     in  bit_2;
         sus2:     in  bit_2;
         sus3:     in  bit_2;
         data_out: out bit_32;
         done:     out bit);
end dfsu;


architecture dfsu_a of dfsu is

    component df4to1mux8
        port (i0: in  bit_8;
              i1: in  bit_8;
              i2: in  bit_8;
              i3: in  bit_8;
              s0: in  bit;
              s1: in  bit;
              o:  out bit_8);
    end component;

begin

    done <= '1' after 1.4 ns when start = '1' else
            '0' after 1 ns;

    mux0: df4to1mux8
        port map(mu_data(7 downto 0), mu_data(15 downto
8),
                 mu_data(23 downto 16), mu_data(31
downto 24),
                 sus0(0), sus0(1), data_out(7 downto
0));

    mux1: df4to1mux8
        port map(mu_data(7 downto 0), mu_data(15 downto
8),
                 mu_data(23 downto 16), mu_data(31
downto 24),
                 sus1(0), sus1(1), data_out(15 downto
8));

    mux2: df4to1mux8
        port map(mu_data(7 downto 0), mu_data(15 downto
8),
                 mu_data(23 downto 16), mu_data(31
downto 24),
                 sus2(0), sus2(1), data_out(23 downto
16));

    mux3: df4to1mux8
        port map(mu_data(7 downto 0), mu_data(15 downto
8),
                 mu_data(23 downto 16), mu_data(31
downto 24),
                 sus3(0), sus3(1), data_out(31 downto
24));

end dfsu_a;
```

DFTRDS

```
library my_packages;
use my_packages.package_1.all;

entity dftrds is
    port(inst: in  bit_32;
         trs0: in  bit;
         trs1: in  bit;
         reg:  out bit_5);
end dftrds;


architecture dftrds_a of dftrds is
begin
    reg <= inst(15 downto 11) after 0.4 ns when
-- special
                          trs0 = '0' and trs1 = '0' else
           inst(20 downto 16) after 0.4 ns when
-- imm or load
                          trs0 = '0' and trs1 = '1' else
           b"11111" after 0.4 ns when
-- jal
                          trs0 = '1' and trs1 = '1' else
           b"00000" after 0.4 ns;
end dftrds_a;
```

DFTSB32

```
library my_packages;
use my_packages.package_1.all;

entity dftsb32 is
    port(i: in bit_32;
         e: in bit;
         o: out bit_32);
end dftsb32;


architecture dftsb32_a of dftsb32 is
begin
    o <= i after 0.3 ns when e = '1' else
         x"0000_0000" after 0.3 ns;
end dftsb32_a;
```

DFWB

```
library my_packages;
use my_packages.package_1.all;

library df_comp;
use df_comp.dfreg32.all;
use df_comp.dfrslat.all;
use df_comp.dfred.all;
use df_comp.dffed.all;
use df_comp.dftrds.all;
use df_comp.df2to1mux.all;
use df_comp.dfreg4.all;

entity dfwb is
    port (wb_start: in  bit;
          inst_in:  in  bit_32;
          data_in:  in  bit_32;
          vbt_in:   in  bit_4;
          wb_data:  out bit_32;
          wb_sel:   out bit_6;
          vbt:      out bit_4;
          wb_done:  out bit);
end dfwb;


architecture dfwb_a of dfwb is

    component dfreg32
        port (d: in  bit_32;
              c: in  bit;
              q: out bit_32);
    end component;

    component dfrslat
        port(s: in  bit;
             r: in  bit;
             q: out bit);
    end component;

    component dfred
        port(i: in  bit;
             o: out bit);
    end component;

    component dffed
        port(i: in  bit;
             o: out bit);
    end component;

    component dftrds
        port(inst: in  bit_32;
             trs0: in  bit;
             trs1: in  bit;
             reg:  out bit_5);
    end component;

    component df2to1mux
        port(i0: in  bit;
             i1: in  bit;
             s:  in  bit;
             o:  out bit);
    end component;
```

```
    component dfreg4
        port (d: in  bit_4;
              c: in  bit;
              q: out bit_4);
    end component;

    signal wb_start_r, wb_start_f, dummy_r, dummy: bit;
    signal trs0, trs1: bit;
    signal instl: bit_32;
    signal reg: bit_5;
    signal moveto_sel, hilo_sel: bit;
    signal count: integer :=0;
begin

    inst_latch: dfreg32
        port map(inst_in, wb_start_r, instl);

    data_latch: dfreg32
        port map(data_in, wb_start_r, wb_data);

    vbt_latch: dfreg4
        port map(vbt_in, wb_start_r, vbt);

    wb_start_red: dfred
        port map(wb_start, wb_start_r);

    wb_start_fed: dffed
        port map(wb_start, wb_start_f);

    done_latch: dfrslat
        port map(dummy_r, wb_start_f, wb_done);

    dummy <= '1' after 8 ns when wb_start'event and
                                 wb_start = '1' else
             '0' after 1 ns;

    dummy_red: dfred
        port map(dummy, dummy_r);

    trds: dftrds
        port map(instl, trs0, trs1, reg);

    trs0 <= '1' after 4.4 ns when -- a23
                                  (instl(31) = '0' and
                                   instl(29 downto 27) =
"001") or
                                  -- a26
                                  (instl(31) = '1' and
                                   instl(29) = '1') or
                                  -- a27
                                  (instl(31) = '0' and
                                   instl(29) = '0' and
                                   instl(27 downto 26) =
"01") or
                                  -- a29
                                  (instl(31) = '0' and
                                   instl(29 downto 28) =
"01") else
            '0' after 4.4 ns;

    trs1 <= '1' after 4.4 ns when -- a21
                                  (instl(31) = '0' and
                                   instl(29 downto 26) =
"0011") or
                                  -- a30
                                  (instl(31) = '1' and
                                   instl(29) = '0') or
                                  -- a31
                                  (instl(31) = '0' and
                                   instl(29) = '1') or
                                  -- a32
                                  (instl(31) = '0' and
                                   instl(29 downto 26) =
"0001" and
                                   instl(20) = '1') else
            '0' after 4.4 ns;

    wb_sel <= moveto_sel & reg(4 downto 1) & hilo_sel;

    moveto_sel <= '1' after 2.6 ns when -- special and
move to instr.
                          instl(31 downto 26) =
"000000" and
                          instl(5 downto 4) = "01" and
                          instl(0) = '1' else
                  '0' after 2.6 ns;

    wb_sel_mux: df2to1mux
        port map(reg(0), instl(1), moveto_sel,
hilo_sel);

    count <= count + 1 when (wb_start'event and
wb_start = '1') else count;

end dfwb_a;
```

APPENDIX D - STRUCTURAL MODEL SOURCE CODE

COMPONENT:	STMIPS
FILENAME:	STMIPS_E.VHDL
DESCRIPTION:	Test bench for structural model of asynchronous version of MIPS R3000 microprocessor (entity)

```
library my_packages;
use my_packages.package_1.all;
use my_packages.dfpack.all;

library df_comp;
use df_comp.dfmemory.all;
use df_comp.dfcompare.all;

library st_comp;
use st_comp.stcpu.all;

entity stmips is
end stmips;
```

COMPONENT:	STMIPS
FILENAME:	STMIPS_A.VHDL
DESCRIPTION:	Test bench for structural model of asynchronous version of MIPS R3000 microprocessor (architecture)

```
architecture stmips_a of stmips is
    component stcpu
        port(sys_control_sig:  in    sys_control_type;
             memory_ack:       in    bit;
             memory_load_ack:  in    question_type;
             compare_ack:      in    question_type;
             compare_load_ack: in    question_type;
             addr_bus:         inout bus_bit_32 bus;
             data_bus:         inout bus_bit_32 bus;
             memory_req:       out   bit;
             memory_w:         out   bit;
             memory_opcode:    out   bit_3;
             memory_load:      out   question_type;
             compare:          out   question_type;
             compare_load:     out   question_type;
             pc_test:          out   bit_32;
             r1_test:          out   bit_32;
             r2_test:          out   bit_32;
             r3_test:          out   bit_32;
             r4_test:          out   bit_32;
             r31_test:         out   bit_32;
             hi_test:          out   bit_32;
             lo_test:          out   bit_32);
    end component;

    component dfmemory
        port(load:     in    question_type;
             req:      in    bit;
             w:        in    bit;
             opcode:   in    bit_3;
             ack:      out   bit;
             load_ack: out   question_type;
             addr_bus: inout bus_bit_32 bus;
             data_bus: inout bus_bit_32 bus);
    end component;

    component dfcompare
        port(compare:          in  question_type;
             compare_load:     in  question_type;
             pc_test:          in  bit_32;
             r1_test:          in  bit_32;
             r2_test:          in  bit_32;
             r3_test:          in  bit_32;
             r4_test:          in  bit_32;
             r31_test:         in  bit_32;
             hi_test:          in  bit_32;
             lo_test:          in  bit_32;
             compare_ack:      out question_type;
             compare_load_ack: out question_type);
    end component;

    signal sys_control: sys_control_type;
    signal memory_load, memory_load_ack: question_type;
    signal memory_req, memory_ack, memory_w: bit;
    signal memory_opcode: bit_3;
    signal addr_bus, data_bus: bus_bit_32;

    signal compare_ack, compare_load_ack:
question_type;
    signal compare, compare_load: question_type;
    signal pc_test, r1_test, r2_test, r3_test: bit_32;
    signal r4_test, r31_test, hi_test, lo_test: bit_32;

begin

    cpu_module: stcpu
        port map(sys_control, memory_ack,
memory_load_ack, compare_ack,
                 compare_load_ack, addr_bus, data_bus,
memory_req,
                 memory_w, memory_opcode, memory_load,
compare,
                 compare_load, pc_test, r1_test,
r2_test, r3_test,
                 r4_test, r31_test, hi_test, lo_test);

    memory_module: dfmemory
        port map(memory_load, memory_req, memory_w,
memory_opcode,
                 memory_ack, memory_load_ack, addr_bus,
data_bus);

    compare_module: dfcompare
        port map(compare, compare_load, pc_test,
r1_test, r2_test,
                 r3_test, r4_test, r31_test, hi_test,
lo_test,
                 compare_ack, compare_load_ack);

end stmips_a;
```

COMPONENT:	STPACK
FILENAME:	STPACK_H.VHDL
DESCRIPTION:	Structural model package (header)

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_1164_extensions.all;

package stpack is

    subtype slv_2  is std_logic_vector (1 downto 0);
    subtype slv_3  is std_logic_vector (2 downto 0);
    subtype slv_4  is std_logic_vector (3 downto 0);
    subtype slv_5  is std_logic_vector (4 downto 0);
    subtype slv_6  is std_logic_vector (5 downto 0);
    subtype slv_8  is std_logic_vector (7 downto 0);
    subtype slv_10 is std_logic_vector (9 downto 0);
    subtype slv_16 is std_logic_vector (15 downto 0);
    subtype slv_20 is std_logic_vector (19 downto 0);
    subtype slv_24 is std_logic_vector (23 downto 0);
    subtype slv_26 is std_logic_vector (25 downto 0);
    subtype slv_30 is std_logic_vector (29 downto 0);
    subtype slv_32 is std_logic_vector (31 downto 0);
    subtype slv_64 is std_logic_vector (63 downto 0);

end stpack;
```

COMPONENT:	STRUCTURAL COMPONENTS
FILENAME:	???_E.VHDL for entity and ???_A.VHDL for architecture
DESCRIPTION:	All components used in structrual model follow (entity shown first)

STALU32

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_1164_extensions.all;

library my_packages;
use my_packages.stpack.all;

library st_comp;
use st_comp.stalu4.all;
use st_comp.stcgen.all;

entity stalu32 is
    port(a:    in  slv_32;
         b:    in  slv_32;
         s0:   in  std_logic;
         s1:   in  std_logic;
         s2:   in  std_logic;
         s3:   in  std_logic;
         cout: out std_logic;
         o:    out slv_32);
end stalu32;


architecture stalu32_a of stalu32 is

    component stalu4
        port(a0:   in  std_logic;
             a1:   in  std_logic;
             a2:   in  std_logic;
             a3:   in  std_logic;
             b0:   in  std_logic;
             b1:   in  std_logic;
             b2:   in  std_logic;
             b3:   in  std_logic;
             cin:  in  std_logic;
             s0:   in  std_logic;
             s1:   in  std_logic;
             s2:   in  std_logic;
             s3:   in  std_logic;
             f0:   out std_logic;
             f1:   out std_logic;
             f2:   out std_logic;
             f3:   out std_logic;
             g:    out std_logic;
             p:    out std_logic;
             cout: out std_logic);
    end component;

    component stcgen
        port(c:  in  std_logic;
             g0: in  std_logic;
             p0: in  std_logic;
             g1: in  std_logic;
             p1: in  std_logic;
             g2: in  std_logic;
             p2: in  std_logic;
             c1: out std_logic;
             c2: out std_logic);
    end component;

    signal cin: std_logic;
    signal g0, g1, g2, g3, g4, g5, g6, g7: std_logic;
    signal p0, p1, p2, p3, p4, p5, p6, p7: std_logic;
    signal cout0, cout1, cout2, cout3, cout4, cout5,
cout6: std_logic;
    signal cgen1_c1, cgen1_c2, cgen2_c1, cgen2_c2:
std_logic;

begin

    alu0to3: stalu4
        port map(a(0), a(1), a(2), a(3), b(0), b(1),
b(2), b(3),
                 cin, s0, s1, s2, s3, o(0), o(1), o(2),
o(3),
                 g0, p0, cout0);

    alu4to7: stalu4
        port map(a(4), a(5), a(6), a(7), b(4), b(5),
b(6), b(7),
                 cout0, s0, s1, s2, s3, o(4), o(5),
o(6), o(7),
                 g1, p1, cout1);

    alu8to11: stalu4
        port map(a(8), a(9), a(10), a(11), b(8), b(9),
b(10), b(11),
                 cgen1_c1, s0, s1, s2, s3, o(8), o(9),
o(10), o(11),
                 g2, p2, cout2);

    alu12to15: stalu4
        port map(a(12), a(13), a(14), a(15), b(12),
b(13), b(14), b(15),
                 cgen1_c2, s0, s1, s2, s3, o(12),
o(13), o(14), o(15),
                 g3, p3, cout3);

    alu16to19: stalu4
        port map(a(16), a(17), a(18), a(19), b(16),
b(17), b(18), b(19),
                 cout3, s0, s1, s2, s3, o(16), o(17),
o(18), o(19),
                 g4, p4, cout4);

    alu20to23: stalu4
        port map(a(20), a(21), a(22), a(23), b(20),
b(21), b(22), b(23),
                 cgen2_c1, s0, s1, s2, s3, o(20),
o(21), o(22), o(23),
                 g5, p5, cout5);

    alu24to27: stalu4
        port map(a(24), a(25), a(26), a(27), b(24),
b(25), b(26), b(27),
                 cgen2_c2, s0, s1, s2, s3, o(24),
o(25), o(26), o(27),
                 g6, p6, cout6);

    alu28to31: stalu4
        port map(a(28), a(29), a(30), a(31), b(28),
b(29), b(30), b(31),
                 cout6, s0, s1, s2, s3, o(28), o(29),
o(30), o(31),
                 g7, p7, cout);

    cgen1: stcgen
        port map(cin, g0, p0, g1, p1, g2, p2, cgen1_c1,
cgen1_c2);

    cgen2: stcgen
        port map(cgen1_c2, g3, p3, g4, p4, g5, p5,
cgen2_c1, cgen2_c2);

end stalu32_a;
```

STALU4

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_1164_extensions.all;

library my_packages;
use my_packages.stpack.all;

entity stalu4 is
    port(a0:   in  std_logic;
         a1:   in  std_logic;
         a2:   in  std_logic;
         a3:   in  std_logic;
         b0:   in  std_logic;
         b1:   in  std_logic;
         b2:   in  std_logic;
         b3:   in  std_logic;
         cin:  in  std_logic;
         s0:   in  std_logic;
         s1:   in  std_logic;
         s2:   in  std_logic;
         s3:   in  std_logic;
         f0:   out std_logic;
         f1:   out std_logic;
         f2:   out std_logic;
         f3:   out std_logic;
         g:    out std_logic;
         p:    out std_logic;
         cout: out std_logic);
end stalu4;


architecture stalu4_a of stalu4 is

    signal n1, n2, n3, n4, n5, n6, n7, n8, n9, n10:
std_logic;
    signal n11, n12, n13, n14, n15, n16, n17, n18, n19,
n20: std_logic;
    signal n21, n22, n23, n24, n25, n26, n27, n28, n29,
n30: std_logic;
    signal n31, n32, n33, n34, n35, n36, n37, n38, n39,
n40: std_logic;
    signal n41, n42, n43, n44, n45, n46, n47, n48, n49,
n50: std_logic;
```

```
    signal n51, n52, n53, n54, n55, n56, n57, n58, n59,
n60: std_logic;
    signal n61, n62, n63, n64, n65, n66, n67, n68, n69,
n70: std_logic;
    signal n71, n72, n73, n74, n75, n76, n77, n78, n79,
n80: std_logic;
    signal n81, n82, n83, n84, n85, g_temp: std_logic;

begin

    n1  <= b0 xor s3 after 2 ns;
    n2  <= a0 xor s3 after 2 ns;
    n3  <= b1 xor s3 after 2 ns;
    n4  <= a1 xor s3 after 2 ns;
    n5  <= b2 xor s3 after 2 ns;
    n6  <= a2 xor s3 after 2 ns;
    n7  <= b3 xor s3 after 2 ns;
    n8  <= a3 xor s3 after 2 ns;
    n9  <= not n1 after 0.3 ns;
    n10 <= not n2 after 0.3 ns;
    n11 <= not n3 after 0.3 ns;
    n12 <= not n4 after 0.3 ns;
    n13 <= not n5 after 0.3 ns;
    n14 <= not n6 after 0.3 ns;
    n15 <= not n7 after 0.3 ns;
    n16 <= not n8 after 0.3 ns;
    n17 <= not s0 after 0.3 ns;
    n18 <= not s1 after 0.3 ns;
    n19 <= not s2 after 0.3 ns;
    n20 <= not n17 after 0.3 ns;
    n21 <= not n18 after 0.3 ns;
    n22 <= not n19 after 0.3 ns;
    n23 <= n17 nand n18 after 0.7 ns;
    n24 <= n18 nand n19 after 0.7 ns;
    n25 <= n17 nand n21 after 0.7 ns;
    n26 <= not (n19 and n20 and n21) after 0.7 ns;
    n27 <= n18 and n22 after 1 ns;
    n28 <= n17 and n22 after 1 ns;
    n29 <= n27 nor n28 after 1 ns;
    n30 <= n9 and n10 and n24 and n25 after 1 ns;
    n31 <= n2 and n9 and n23 and n25 and n26 after 1
ns;
    n32 <= n1 and n10 and n24 and n26 after 1 ns;
    n33 <= n9 and n10 and n23 and n26 and n29 after 1
ns;
    n34 <= n2 and n9 and n24 and n25 after 1 ns;
    n35 <= n1 and n10 and n24 and n25 after 1 ns;
    n36 <= n1 and n2 and n23 and n26 after 1 ns;
    n37 <= n11 and n12 and n24 and n25 after 1 ns;
    n38 <= n4 and n11 and n22 and n25 and n26 after 1
ns;
    n39 <= n3 and n12 and n24 and n26 after 1 ns;
    n40 <= n11 and n12 and n23 and n26 and n29 after 1
ns;
    n41 <= n4 and n11 and n24 and n25 after 1 ns;
    n42 <= n3 and n12 and n24 and n25 after 1 ns;
    n43 <= n3 and n4 and n23 and n26 after 1 ns;
    n44 <= n13 and n14 and n24 and n25 after 1 ns;
    n45 <= n6 and n13 and n23 and n25 and n26 after 1
ns;
    n46 <= n5 and n14 and n24 and n26 after 1 ns;
    n47 <= n13 and n14 and n23 and n26 and n29 after 1
ns;
    n48 <= n6 and n13 and n24 and n25 after 1 ns;
    n49 <= n5 and n14 and n24 and n25 after 1 ns;
    n50 <= n5 and n6 and n23 and n26 after 1 ns;
    n51 <= n15 and n16 and n24 and n25 after 1 ns;
    n52 <= n8 and n15 and n23 and n25 and n26 after 1
ns;
    n53 <= n7 and n16 and n24 and n26 after 1 ns;
    n54 <= n15 and n16 and n23 and n26 and n29 after 1
ns;
    n55 <= n8 and n15 and n24 and n25 after 1 ns;
    n56 <= n7 and n16 and n24 and n25 after 1 ns;
    n57 <= n7 and n8 and n23 and n26 after 1 ns;
    n58 <= n17 and n18 after 1 ns;
    n59 <= not (n30 or n31 or n32) after 1 ns;
    n60 <= not (n33 or n34 or n35 or n36) after 1 ns;
    n61 <= not (n37 or n38 or n39) after 1 ns;
    n62 <= not (n40 or n41 or n42 or n43) after 1 ns;
    n63 <= not (n44 or n45 or n46) after 1 ns;
    n64 <= not (n47 or n48 or n49 or n50) after 1 ns;
    n65 <= not (n51 or n52 or n53) after 1 ns;
    n66 <= not (n54 or n55 or n56 or n57) after 1 ns;
    n67 <= n22 nor n58 after 1 ns;
    n68 <= cin and n59 and n67 after 1 ns;
    n69 <= n59 and n60 and n67 after 1 ns;
    n70 <= cin and n59 and n61 and n67 after 1 ns;
    n71 <= n59 and n60 and n61 and n67 after 1 ns;
    n72 <= n61 and n62 and n67 after 1 ns;
    n73 <= cin and n59 and n61 and n63 and n67 after 1
ns;
    n74 <= n59 and n60 and n61 and n63 and n67 after 1
ns;
    n75 <= n61 and n62 and n63 and n67 after 1 ns;
    n76 <= n63 and n64 and n67 after 1 ns;
    n77 <= cin nand n67 after 0.7 ns;
    n78 <= n68 nor n69 after 1 ns;
    n79 <= not (n70 or n71 or n72) after 1 ns;
    n80 <= not (n73 or n74 or n75 or n76) after 1 ns;
    n81 <= n59 and n60 and n61 and n63 and n65 after 1
ns;
    n82 <= n61 and n62 and n63 and n65 after 1 ns;
    n83 <= n63 and n64 and n65 after 1 ns;
    n84 <= n65 and n66 after 1 ns;
    n85 <= not (cin and n59 and n61 and n63 and n65)
after 0.7 ns;
    f0 <= n60 xor n77 after 2 ns;
    f1 <= n62 xor n78 after 2 ns;
    f2 <= n64 xor n79 after 2 ns;
    f3 <= n66 xor n80 after 2 ns;
    p <= not (n59 and n61 and n63 and n65) after 0.7
ns;
    g_temp <= not (n81 or n82 or n83 or n84) after 1
ns;
    g <= g_temp;
    cout <= g_temp nand n85 after 0.7 ns;

end stalu4_a;
```

STALU

```
library mgc_portable;
use mgc_portable.qsim_tc.all;

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_1164_extensions.all;

library my_packages;
use my_packages.package_1.all;
use my_packages.stpack.all;

library df_comp;
use df_comp.dfreg32.all;
use df_comp.dfregr.all;
use df_comp.dfred.all;
use df_comp.dffed.all;
use df_comp.dfrslat.all;
use df_comp.df2to1mux32.all;
use df_comp.dfmd.all;
use df_comp.dfhlreg.all;
use df_comp.dfadd8.all;

library st_comp;
use st_comp.stregbank.all;
use st_comp.stalublk.all;

entity stalu is
    port (alu_start:   in  bit;
          ibo:         in  bit;
          alu_select:  in  bit;
          md_select:   in  bit;
          add8_select: in  bit;
          reg1_out:    in  bit_5;
          reg2_out:    in  bit_5;
          treg_out:    in  bit_5;
          pcl:         in  bit_32;
          inst_in:     in  bit_32;
          wb_data:     in  bit_32;
          wb_sel:      in  bit_6;
          vbt:         in  bit_4;
          write:       in  bit;
          jjr:         in  bit;
          set_hi_db:   in  bit;
          set_lo_db:   in  bit;
          cc:          out bit;
          hi_db:       out bit;
          lo_db:       out bit;
          db:          out bit_32;
          reg:         out bit_32;
          alu_exc:     out bit;
          inst_out:    out bit_32;
          data1_out:   out bit_32;
          data2_out:   out bit_32;
          ccd:         out bit;
          r1_test:     out bit_32;
          r2_test:     out bit_32;
          r3_test:     out bit_32;
          r4_test:     out bit_32;
          r31_test:    out bit_32;
          hi_test:     out bit_32;
          lo_test:     out bit_32;
```

```
        alu_done:    out bit);
end stalu;


architecture stalu_a of stalu is

    component dfreg32
        port (d: in  bit_32;
              c: in  bit;
              q: out bit_32);
    end component;

    component dfregr
        port(d: in  bit;
             c: in  bit;
             r: in  bit;
             q: out bit);
    end component;

    component dfred
        port(i: in bit;
             o: out bit);
    end component;

    component dffed
        port(i: in bit;
             o: out bit);
    end component;

    component dfrslat
        port(s: in  bit;
             r: in  bit;
             q: out bit);
    end component;

    component df2to1mux32
        port(i0: in  bit_32;
             i1: in  bit_32;
             s:  in  bit;
             o:  out bit_32);
    end component;

    component stregbank
        port(data:     in  slv_32;
             reg_sel:  in  slv_5;
             a_sel:    in  slv_5;
             b_sel:    in  slv_5;
             db_sel:   in  slv_5;
             write:    in  std_logic;
             start:    in  std_logic;
             vbt:      in  slv_4;
             a:        out slv_32;
             b:        out slv_32;
             db:       out slv_32;
             r1_test:  out slv_32;
             r2_test:  out slv_32;
             r3_test:  out slv_32;
             r4_test:  out slv_32;
             r31_test: out slv_32);
    end component;

    component stalublk
        port(a_in:      in  bit_32;
             b_in:      in  bit_32;
             inst_in:   in  bit_32;
             pcl_in:    in  bit_32;
             ibo:       in  bit;
             start:     in  bit;
             lat_inst:  in  bit;
             hilo:      in  bit_32;
             add8:      in  bit_32;
             exc:       out bit;
             cc:        out bit;
             b_out:     out bit_32;
             output:    out bit_32;
             inst_out:  out bit_32;
             pcl_out:   out bit_32;
             alub_done: out bit);
    end component;

    component dfmd
        port(start: in  bit;
             x:     in  bit_32;
             y:     in  bit_32;
             sign:  in  bit;
             divd:  in  bit;
             hi:    out bit_32;
             lo:    out bit_32;
             done1: out bit;
             done2: out bit);
    end component;

    component dfhlreg
        port(start:     in  bit;
             hi_in:     in  bit_32;
             lo_in:     in  bit_32;
             load_hi:   in  bit;
             load_lo:   in  bit;
             set_hi_db: in  bit;
             set_lo_db: in  bit;
             hi_out:    out bit_32;
             lo_out:    out bit_32;
             hi_db:     out bit;
             lo_db:     out bit);
    end component;

    component dfadd8
        port(i:     in  bit_32;
             start: in  bit;
             o:     out bit_32;
             done:  out bit);
    end component;

signal a, b, hilo, add8_out, add8_in: bit_32;
signal x, y, hi1, hi2, hi3, lo1, lo2, lo3: bit_32;
signal md_inst_l: bit_32;
signal start1, exc, alub_done, alu_select_l: bit;
signal add8_select_l, add8_start, add8_done: bit;
signal alu_done_temp, sign, divd, start2: bit;
signal md_done1, md_done2, md_select_l: bit;
signal load_hi, load_lo, l_hi, l_lo: bit;
signal jjr_done, alu_start_r, alu_start_f: bit;
signal done_temp_r, done_temp: bit;
signal write_nomove: bit;

signal Cwb_data, Ca, Cb, Cdb: slv_32;
signal Cwb_sel, Creg1_out, Creg2_out, Ctreg_out: slv_5;
signal Cwrite_nomove, Calu_start_r: std_logic;
signal Cvbt: slv_4;
signal Cr1_test, Cr2_test, Cr3_test, Cr4_test,
Cr31_test: slv_32;

begin

    alu_start_red: dfred
        port map(alu_start, alu_start_r);

    Cwb_data <= To_StdLogicVector(wb_data);
    Cwb_sel <= To_StdLogicVector(wb_sel(4 downto 0));
    Creg1_out <= To_StdLogicVector(reg1_out);
    Creg2_out <= To_StdLogicVector(reg2_out);
    Ctreg_out <= To_StdLogicVector(treg_out);
    Cwrite_nomove <= To_StdULogic(write_nomove);
    Calu_start_r <= To_StdULogic(alu_start_r);
    Cvbt <= To_StdLogicVector(vbt);

    a <= To_bitvector(Ca);
    b <= To_bitvector(Cb);
    db <= To_bitvector(Cdb);
    r1_test <= To_bitvector(Cr1_test);
    r2_test <= To_bitvector(Cr2_test);
    r3_test <= To_bitvector(Cr3_test);
    r4_test <= To_bitvector(Cr4_test);
    r31_test <= To_bitvector(Cr31_test);

    register_bank: stregbank
        port map(Cwb_data, Cwb_sel, Creg1_out,
Creg2_out,
                 Ctreg_out, Cwrite_nomove,
Calu_start_r, Cvbt, Ca, Cb, Cdb,
                 Cr1_test, Cr2_test, Cr3_test,
Cr4_test, Cr31_test);

    hi_test <= hi3;
    lo_test <= lo3;

    write_nomove <= write and (not wb_sel(5)) after 2
ns;

    reg <= a;

    alu_block: stalublk
        port map(a, b, inst_in, pcl, ibo, start1,
alu_start_r,
                 hilo, add8_out, exc, cc, data2_out,
data1_out,
                 inst_out, add8_in, alub_done);

    alu_select_latch: dfregr
        port map(alu_select, alu_start_r,
alu_done_temp, alu_select_l);

    add8_select_latch: dfregr
        port map(add8_select, alu_start_r,
alu_done_temp, add8_select_l);
```

```
    start1 <= alu_start and alu_select_l after 1 ns;

    add8_start <= alu_start and add8_select_l after 1
ns;

    add8: dfadd8
        port map(add8_in, add8_start, add8_out,
add8_done);

    alu_exc <= exc and alu_done_temp after 1 ns;

    md: dfmd
        port map(start2, x, y, sign, divd, hi1, lo1,
md_done1,
                 md_done2);

    md_x_latch: dfreg32
        port map(a, start2, x);

    md_y_latch: dfreg32
        port map(b, start2, y);

    md_inst_latch: dfreg32
        port map(inst_in, start2, md_inst_l);

    sign <= not md_inst_l(0);

    divd <= md_inst_l(1);

    md_select_latch: dfregr
        port map(md_select, alu_start_r, alu_done_temp,
md_select_l);

    start2 <= alu_start and md_select_l after 1 ns;

    hi_mux: df2to1mux32
        port map(hi1, wb_data, wb_sel(5), hi2);

    lo_mux: df2to1mux32
        port map(lo1, wb_data, wb_sel(5), lo2);

    hilo_reg: dfhlreg
        port map(alu_start, hi2, lo2, load_hi, load_lo,
                 set_hi_db, set_lo_db, hi3, lo3, hi_db,
lo_db);

    load_hi <= l_hi or md_done2 after 1.3 ns;
    load_lo <= l_lo or md_done2 after 1.3 ns;

    l_hi <= write and wb_sel(5) and (not wb_sel(0))
after 1.4 ns;
    l_lo <= write and wb_sel(5) and      wb_sel(0)
after 1.1 ns;

    hilo_mux: df2to1mux32
        port map(hi3, lo3, md_inst_l(1), hilo);

    done_temp <= md_done1 or jjr_done or add8_done or
                 alub_done after 1.5 ns;

    done_temp_red: dfred
        port map(done_temp, done_temp_r);

    alu_start_fed: dffed
        port map(alu_start, alu_start_f);

    done_latch: dfrslat
        port map(done_temp_r, alu_start_f,
alu_done_temp);

    alu_done <= alu_done_temp;

    jjr_latch: dfregr
        port map(jjr, alu_start_r, alu_done_temp,
jjr_done);

    ccd_latch: dfrslat
        port map(done_temp_r, alu_start_r, ccd);

end stalu_a;
```

STALUBLK

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_1164_extensions.all;

library my_packages;
use my_packages.package_1.all;
use my_packages.stpack.all;

library df_comp;
use df_comp.dfbusctl.all;
use df_comp.dfaludec.all;
use df_comp.dfovrf.all;
use df_comp.dfcomp.all;
use df_comp.dfbctl.all;
use df_comp.dfsctl.all;
use df_comp.dfshift.all;
use df_comp.dfslt.all;
use df_comp.dfoutsel.all;

library st_comp;
use st_comp.stalu32.all;

entity stalublk is
    port(a_in:      in  bit_32;
         b_in:      in  bit_32;
         inst_in:   in  bit_32;
         pcl_in:    in  bit_32;
         ibo:       in  bit;
         start:     in  bit;
         lat_inst:  in  bit;
         hilo:      in  bit_32;
         add8:      in  bit_32;
         exc:       out bit;
         cc:        out bit;
         b_out:     out bit_32;
         output:    out bit_32;
         inst_out:  out bit_32;
         pcl_out:   out bit_32;
         alub_done: out bit);
end stalublk;


architecture stalublk_a of stalublk is

component dfbusctl
    port(inst_in:   in  bit_32;
         a_in:      in  bit_32;
         b_in:      in  bit_32;
         pcl_in:    in  bit_32;
         start:     in  bit;
         lat_inst:  in  bit;
         ibo:       in  bit;
         inst_out:  out bit_32;
         a_out:     out bit_32;
         b_out:     out bit_32;
         a:         out bit_5;
         b_pass:    out bit_32;
         pcl_out:   out bit_32;
         alub_done: out bit);
end component;

component dfaludec
    port(i:  in  bit_32;
         s0: out bit;
         s1: out bit;
         s2: out bit;
         s3: out bit);
end component;

component stalu32
    port(a:     in  slv_32;
         b:     in  slv_32;
         s0:    in  std_logic;
         s1:    in  std_logic;
         s2:    in  std_logic;
         s3:    in  std_logic;
         cout:  out std_logic;
         o:     out slv_32);
end component;

component dfovrf
    port(inst:     in  bit_32;
         data:     in  bit_32;
         carry:    in  bit;
         overflow: out bit);
end component;

component dfcomp
    port(i:   in  bit_32;
         ltz: out bit;
         eqz: out bit;
         gtz: out bit;
         o:   out bit_32);
end component;

component dfbctl
    port(inst: in  bit_32;
         ltz:  in  bit;
         eqz:  in  bit;
         gtz:  in  bit;
         cc:   out bit);
```

```
end component;

component dfsctl
    port(inst: in  bit_32;
         a:    in  bit_5;
         lr:   out bit;
         la:   out bit;
         sel:  out bit_5);
end component;

component dfshift
    port(i:   in  bit_32;
         lr:  in  bit;
         la:  in  bit;
         sel: in  bit_5;
         o:   out bit_32);
end component;

component dfslt
    port(i:    in  bit_32;
         inst: in  bit_32;
         ltz:  in  bit;
         o:    out bit_32);
end component;

component dfoutsel
    port(inst:   in  bit_32;
         alu:    in  bit_32;
         add8:   in  bit_32;
         hilo:   in  bit_32;
         output: out bit_32);
end component;

signal inst1, a_line, b_line, alu_out, comp_out:
bit_32;
signal shift_out, slt_out: bit_32;
signal a: bit_5;
signal s0, s1, s2, s3, c_out, ltz, eqz, gtz, lr, la:
bit;
signal sel: bit_5;

signal Ca_line, Cb_line, Calu_out: slv_32;
signal Cs0, Cs1, Cs2, Cs3, Ccout: std_logic;

begin

    bus_control: dfbusctl
        port map(inst_in, a_in, b_in, pc1_in, start,
                 lat_inst, ibo, inst1, a_line, b_line,
                 a, b_out, pc1_out, alub_done);

    inst_out <= inst1;

    alu_decode: dfaludec
        port map(inst1, s0, s1, s2, s3);

    Ca_line <= To_StdLogicVector(a_line);
    Cb_line <= To_StdLogicVector(b_line);
    Cs0 <= To_StdULogic(s0);
    Cs1 <= To_StdULogic(s1);
    Cs2 <= To_StdULogic(s2);
    Cs3 <= To_StdULogic(s3);
    c_out <= To_bit(Ccout);
    alu_out <= To_bitvector(Calu_out);

    alu: stalu32
        port map(Ca_line, Cb_line, Cs0, Cs1,
                 Cs2, Cs3, Ccout, Calu_out);
    ovrf: dfovrf
        port map(inst1, alu_out, c_out, exc);

    compare: dfcomp
        port map(alu_out, ltz, eqz, gtz, comp_out);

    branch_ctl: dfbctl
        port map(inst1, ltz, eqz, gtz, cc);

    shift_ctl: dfsctl
        port map(inst1, a, lr, la, sel);

    shifter: dfshift
        port map(comp_out, lr, la, sel, shift_out);

    slt_box: dfslt
        port map(shift_out, inst1, ltz, slt_out);

    output_selector: dfoutsel
        port map(inst1, slt_out, add8, hilo, output);

end stalublk_a;
```

STCGEN

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_1164_extensions.all;

library my_packages;
use my_packages.stpack.all;

entity stcgen is
    port(c:  in  std_logic;
         g0: in  std_logic;
         p0: in  std_logic;
         g1: in  std_logic;
         p1: in  std_logic;
         g2: in  std_logic;
         p2: in  std_logic;
         c1: out std_logic;
         c2: out std_logic);
end stcgen;


architecture stcgen_a of stcgen is

    signal a, b, d, e, f, g, h, c_bar: std_logic;

begin

    c_bar <= not c after 0.3 ns;
    a <= g2 and g1 and g0 and c_bar after 1 ns;
    b <= p0 and g2 and g1 and g0 after 1 ns;
    d <= p1 and g2 and g1 after 1 ns;
    e <= p2 and g2 after 1 ns;
    f <= g1 and g0 and c_bar after 1 ns;
    g <= p0 and g1 and g0 after 1 ns;
    h <= p1 and g1 after 1 ns;
    c2 <= a or b or d or e after 1.3 ns;
    c1 <= f or g or h after 1.3 ns;

end stcgen_a;
```

STCPU

```
library my_packages;
use my_packages.package_1.all;

library df_comp;
use df_comp.dfhcc.all;
use df_comp.dfif.all;
use df_comp.dfid.all;
use df_comp.dfmem.all;
use df_comp.dfwb.all;
use df_comp.dfeh.all;
use df_comp.dfbc.all;
use df_comp.dfrslat.all;
use df_comp.dfreg32.all;

library st_comp;
use st_comp.stalu.all;

entity stcpu is
    port(sys_control_sig:  in    sys_control_type;
         memory_ack:       in    bit;
         memory_load_ack:  in    question_type;
         compare_ack:      in    question_type;
         compare_load_ack: in    question_type;
         addr_bus:         inout bus_bit_32 bus;
         data_bus:         inout bus_bit_32 bus;
         memory_req:       out   bit;
         memory_w:         out   bit;
         memory_opcode:    out   bit_3;
         memory_load:      out   question_type;
         compare:          out   question_type;
         compare_load:     out   question_type;
         pc_test:          out   bit_32;
         r1_test:          out   bit_32;
         r2_test:          out   bit_32;
         r3_test:          out   bit_32;
         r4_test:          out   bit_32;
         r31_test:         out   bit_32;
         hi_test:          out   bit_32;
         lo_test:          out   bit_32);
end stcpu;


architecture stcpu_a of stcpu is

    -- component declarations
    component dfhcc
        port (init:    in  bit;
```

```
        prev_ok: in  bit;
        ready:   in  bit;
        ain:     in  bit;
        aout:    out bit;
        rout:    out bit;
        ok:      out bit);
end component;

component dfif
    port (if_start:   in    bit;
          ia:         in    bit;
          int_req:    in    bit;
          addr_valid: in    bit;
          iv:         in    bit_32;
          new_pc:     in    bit_32;
          addr_bus:   inout bus_bit_32;
          data_bus:   inout bus_bit_32;
          ir:         out   bit;
          il:         out   bit;
          inst:       out   bit_32;
          pc:         out   bit_32;
          if_exc:     out   bit;
          if_done:    out   bit);
end component;

component dfid
    port (id_start:    in  bit;
          hi_db:       in  bit;
          lo_db:       in  bit;
          db:          in  bit_32;
          inst_in:     in  bit_32;
          cc:          in  bit;
          ccd:         in  bit;
          pc:          in  bit_32;
          ir:          in  bit;
          reg:         in  bit_32;
          illegal:     out bit;
          id_exc:      out bit;
          ibo:         out bit;
          alu_select:  out bit;
          md_select:   out bit;
          add8_select: out bit;
          reg1_out:    out bit_5;
          reg2_out:    out bit_5;
          treg_out:    out bit_5;
          addr_valid:  out bit;
          new_pc:      out bit_32;
          pcl:         out bit_32;
          inst_out:    out bit_32;
          set_hi_db:   out bit;
          set_lo_db:   out bit;
          jjr:         out bit;
          id_done:     out bit);
end component;

component stalu
    port (alu_start:   in  bit;
          ibo:         in  bit;
          alu_select:  in  bit;
          md_select:   in  bit;
          add8_select: in  bit;
          reg1_out:    in  bit_5;
          reg2_out:    in  bit_5;
          treg_out:    in  bit_5;
          pcl:         in  bit_32;
          inst_in:     in  bit_32;
          wb_data:     in  bit_32;
          wb_sel:      in  bit_6;
          vbt:         in  bit_4;
          write:       in  bit;
          jjr:         in  bit;
          set_hi_db:   in  bit;
          set_lo_db:   in  bit;
          co:          out bit;
          hi_db:       out bit;
          lo_db:       out bit;
          db:          out bit_32;
          reg:         out bit_32;
          alu_exc:     out bit;
          inst_out:    out bit_32;
          data1_out:   out bit_32;
          data2_out:   out bit_32;
          ccd:         out bit;
          r1_test:     out bit_32;
          r2_test:     out bit_32;
          r3_test:     out bit_32;
          r4_test:     out bit_32;
          r31_test:    out bit_32;
          hi_test:     out bit_32;
          lo_test:     out bit_32;
          alu_done:    out bit);
end component;

component dfmem
    port (mem_start: in    bit;
          ma:        in    bit;
          inst_in:   in    bit_32;
          data1_in:  in    bit_32;
          data2_in:  in    bit_32;
          addr_bus:  inout bus_bit_32;
          data_bus:  inout bus_bit_32;
          mem_exc:   out   bit;
          mr:        out   bit;
          ml:        out   bit;
          w:         out   bit;
          opcode:    out   bit_3;
          inst_out:  out   bit_32;
          data_out:  out   bit_32;
          vbt:       out   bit_4;
          mem_done:  out   bit);
end component;

component dfwb
    port (wb_start: in  bit;
          inst_in:  in  bit_32;
          data_in:  in  bit_32;
          vbt_in:   in  bit_4;
          wb_data:  out bit_32;
          wb_sel:   out bit_6;
          vbt:      out bit_4;
          wb_done:  out bit);
end component;

component dfeh
    port (illegal:    in  bit;
          if_exc:     in  bit;
          id_exc:     in  bit;
          alu_exc:    in  bit;
          mem_exc:    in  bit;
          int_req:    out bit;
          int_vector: out bit_32);
end component;

component dfbc
    port (ir:  in  bit;
          il:  in  bit;
          mr:  in  bit;
          ml:  in  bit;
          ack: in  bit;
          ia:  out bit;
          ma:  out bit;
          req: out bit);
end component;

component dfrslat
    port(s: in  bit;
         r: in  bit;
         q: out bit);
end component;

component dfreg32
    port (d: in  bit_32;
          c: in  bit;
          q: out bit_32);
end component;


-- system control
signal start: bit;

signal begin_in: bit;

-- pipeline handshake
signal begin_ok, if_ok, id_ok, alu_ok, mem_ok,
wb_ok: bit := '1';
signal if_ack, id_ack, alu_ack, mem_ack, wb_ack,
end_ack: bit;
signal if_start, id_start, alu_start, mem_start,
wb_start: bit;
signal if_done, id_done, alu_done, mem_done,
wb_done: bit;
signal init: bit;

-- interrupt/exception control
signal if_exc, id_exc, alu_exc, mem_exc: bit;
signal illegal, int_req: bit;
signal int_vector: bit_32;
signal int_latch_q: bit;

constant zero_constant: bit := '0';
signal int_reset: bit := zero_constant;

-- bus control
signal if_bus_req, mem_bus_req: bit;
signal if_bus_ack, mem_bus_ack: bit;
signal if_load_addr, mem_load_addr: bit;
```

```
    -- data, control
    signal inst_ifid, pc, new_pc: bit_32;
    signal addr_valid: bit;

    signal cc, hi_db, lo_db: bit;
    signal db, reg: bit_32;
    signal ibo, alu_select, md_select, add8_select:
bit;
    signal reg1_out, reg2_out, treg_out: bit_5;
    signal pcl, inst_idalu: bit_32;
    signal set_hi_db, set_lo_db, jjr: bit;

    signal wb_data, data1_out, data2_out, inst_alumem:
bit_32;
    signal wb_sel: bit_6;
    signal vbt: bit_4;

    signal inst_memwb, data_memwb: bit_32;
    signal vbt_memwb: bit_4;

    signal ccd: bit;

    -- signals for test bench
    signal pc_alumem, pc_memwb: bit_32;
    signal test_mode, wb_done2: bit;
begin

    system: process
    begin
        wait on sys_control_sig;
        case sys_control_sig is
          when stop =>
            start <= '0';

          when reset =>
            null;

          when load =>
            memory_load <= yes after delay;
            wait until memory_load_ack = yes;

            if test_mode = '1' then
                compare_load <= yes after delay;
                wait until compare_load_ack = yes;
            end if;

          when run =>
            start <= '1';
        end case;
    end process system;

    int_latch: dfrslat
        port map(int_req, int_reset, int_latch_q);

    bus_control: dfbc
        port map(if_bus_req, if_load_addr, mem_bus_req,
mem_load_addr,
                 memory_ack, if_bus_ack, mem_bus_ack,
memory_req);

    exception_handler: dfeh
        port map(illegal, if_exc, id_exc, alu_exc,
                 mem_exc, int_req, int_vector);

    begin_in <= not (if_ack or start) after 1 ns;

    begin_ok <= begin_in or int_latch_q after 1.3 ns;

    if_hcc: dfhcc
        port map(init, begin_ok, if_done, id_ack,
                 if_ack, if_start, if_ok);

    id_hcc: dfhcc
        port map(init, if_ok, id_done, alu_ack,
                 id_ack, id_start, id_ok);

    alu_hcc: dfhcc
        port map(init, id_ok, alu_done, mem_ack,
                 alu_ack, alu_start, alu_ok);

    mem_hcc: dfhcc
        port map(init, alu_ok, mem_done, wb_ack,
                 mem_ack, mem_start, mem_ok);

    wb_hcc: dfhcc
        port map(init, mem_ok, wb_done2, end_ack,
                 wb_ack, wb_start, wb_ok);

    end_ack  <= not wb_ok after 0.3 ns;


    if_stage: dfif
        port map(if_start, if_bus_ack, int_req,
addr_valid,
                 int_vector, new_pc, addr_bus,
data_bus,if_bus_req,
                 if_load_addr, inst_ifid, pc, if_exc,
if_done);

    id_stage: dfid
        port map(id_start, hi_db, lo_db, db, inst_ifid,
cc, ccd,
                 pc, if_bus_req, reg, illegal, id_exc,
ibo, alu_select,
                 md_select, add8_select, reg1_out,
reg2_out, treg_out,
                 addr_valid, new_pc, pcl, inst_idalu,
set_hi_db,
                 set_lo_db, jjr, id_done);

    alu_stage: stalu
        port map(alu_start, ibo, alu_select, md_select,
add8_select,
                 reg1_out, reg2_out, treg_out, pcl,
inst_idalu, wb_data,
                 wb_sel, vbt, wb_done, jjr, set_hi_db,
set_lo_db,
                 cc, hi_db, lo_db, db, reg, alu_exc,
inst_alumem,
                 data1_out, data2_out, ccd, r1_test,
r2_test, r3_test,
                 r4_test, r31_test, hi_test, lo_test,
alu_done);

    mem_stage: dfmem
        port map(mem_start, mem_bus_ack, inst_alumem,
data1_out,
                 data2_out, addr_bus, data_bus,
mem_exc, mem_bus_req,
                 mem_load_addr, memory_w,
memory_opcode, inst_memwb,
                 data_memwb, vbt_memwb, mem_done);

    wb_stage: dfwb
        port map(wb_start, inst_memwb, data_memwb,
vbt_memwb,
                 wb_data, wb_sel, vbt, wb_done);

    alu_pc_latch: dfreg32
        port map(pcl, alu_start, pc_alumem);

    mem_pc_latch: dfreg32
        port map(pc_alumem, mem_start, pc_memwb);

    wb_pc_latch: dfreg32
        port map(pc_memwb, wb_start, pc_test);

    hcc_tb_control: process
    begin
        wait on wb_done;

        if test_mode = '0' then
            if wb_done = '1' then
                wb_done2 <= '1';
            else
                wb_done2 <= '0';
            end if;
        else
            if wb_done = '1' then
                compare <= yes after delay;
                wait until compare_ack = yes;
                compare <= no after 1 ns;
                wait until compare_ack = no;
                wb_done2 <= '1';
            else
                wb_done2 <= '0';
            end if;
        end if;
    end process hcc_tb_control;

end stcpu_a;
```

STDEC32

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_1164_extensions.all;

library my_packages;
use my_packages.stpack.all;

entity stdec32 is
    port(i:   in  slv_5;
```

```
        o0:  out std_logic;
        o1:  out std_logic;
        o2:  out std_logic;
        o3:  out std_logic;
        o4:  out std_logic;
        o5:  out std_logic;
        o6:  out std_logic;
        o7:  out std_logic;
        o8:  out std_logic;
        o9:  out std_logic;
        o10: out std_logic;
        o11: out std_logic;
        o12: out std_logic;
        o13: out std_logic;
        o14: out std_logic;
        o15: out std_logic;
        o16: out std_logic;
        o17: out std_logic;
        o18: out std_logic;
        o19: out std_logic;
        o20: out std_logic;
        o21: out std_logic;
        o22: out std_logic;
        o23: out std_logic;
        o24: out std_logic;
        o25: out std_logic;
        o26: out std_logic;
        o27: out std_logic;
        o28: out std_logic;
        o29: out std_logic;
        o30: out std_logic;
        o31: out std_logic);
end stdec32;

architecture stdec32_a of stdec32 is
begin
    o0  <= '1' after 0.3 ns when i = "00000" else '0';
    o1  <= '1' after 0.3 ns when i = "00001" else '0';
    o2  <= '1' after 0.3 ns when i = "00010" else '0';
    o3  <= '1' after 0.3 ns when i = "00011" else '0';
    o4  <= '1' after 0.3 ns when i = "00100" else '0';
    o5  <= '1' after 0.3 ns when i = "00101" else '0';
    o6  <= '1' after 0.3 ns when i = "00110" else '0';
    o7  <= '1' after 0.3 ns when i = "00111" else '0';
    o8  <= '1' after 0.3 ns when i = "01000" else '0';
    o9  <= '1' after 0.3 ns when i = "01001" else '0';
    o10 <= '1' after 0.3 ns when i = "01010" else '0';
    o11 <= '1' after 0.3 ns when i = "01011" else '0';
    o12 <= '1' after 0.3 ns when i = "01100" else '0';
    o13 <= '1' after 0.3 ns when i = "01101" else '0';
    o14 <= '1' after 0.3 ns when i = "01110" else '0';
    o15 <= '1' after 0.3 ns when i = "01111" else '0';
    o16 <= '1' after 0.3 ns when i = "10000" else '0';
    o17 <= '1' after 0.3 ns when i = "10001" else '0';
    o18 <= '1' after 0.3 ns when i = "10010" else '0';
    o19 <= '1' after 0.3 ns when i = "10011" else '0';
    o20 <= '1' after 0.3 ns when i = "10100" else '0';
    o21 <= '1' after 0.3 ns when i = "10101" else '0';
    o22 <= '1' after 0.3 ns when i = "10110" else '0';
    o23 <= '1' after 0.3 ns when i = "10111" else '0';
    o24 <= '1' after 0.3 ns when i = "11000" else '0';
    o25 <= '1' after 0.3 ns when i = "11001" else '0';
    o26 <= '1' after 0.3 ns when i = "11010" else '0';
    o27 <= '1' after 0.3 ns when i = "11011" else '0';
    o28 <= '1' after 0.3 ns when i = "11100" else '0';
    o29 <= '1' after 0.3 ns when i = "11101" else '0';
    o30 <= '1' after 0.3 ns when i = "11110" else '0';
    o31 <= '1' after 0.3 ns when i = "11111" else '0';
end stdec32_a;
```

STDECE32

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_1164_extensions.all;

library my_packages;
use my_packages.stpack.all;

entity stdece32 is
    port(i:   in  slv_5;
         e:   in  std_logic;
         o0:  out std_logic;
         o1:  out std_logic;
         o2:  out std_logic;
         o3:  out std_logic;
         o4:  out std_logic;
         o5:  out std_logic;
         o6:  out std_logic;
         o7:  out std_logic;
         o8:  out std_logic;
         o9:  out std_logic;
         o10: out std_logic;
         o11: out std_logic;
         o12: out std_logic;
         o13: out std_logic;
         o14: out std_logic;
         o15: out std_logic;
         o16: out std_logic;
         o17: out std_logic;
         o18: out std_logic;
         o19: out std_logic;
         o20: out std_logic;
         o21: out std_logic;
         o22: out std_logic;
         o23: out std_logic;
         o24: out std_logic;
         o25: out std_logic;
         o26: out std_logic;
         o27: out std_logic;
         o28: out std_logic;
         o29: out std_logic;
         o30: out std_logic;
         o31: out std_logic);
end stdece32;

architecture stdece32_a of stdece32 is
begin
    o0  <= '1' after 0.3 ns when i = "00000" and
e'event and e = '1' else '0';
    o1  <= '1' after 0.3 ns when i = "00001" and
e'event and e = '1' else '0';
    o2  <= '1' after 0.3 ns when i = "00010" and
e'event and e = '1' else '0';
    o3  <= '1' after 0.3 ns when i = "00011" and
e'event and e = '1' else '0';
    o4  <= '1' after 0.3 ns when i = "00100" and
e'event and e = '1' else '0';
    o5  <= '1' after 0.3 ns when i = "00101" and
e'event and e = '1' else '0';
    o6  <= '1' after 0.3 ns when i = "00110" and
e'event and e = '1' else '0';
    o7  <= '1' after 0.3 ns when i = "00111" and
e'event and e = '1' else '0';
    o8  <= '1' after 0.3 ns when i = "01000" and
e'event and e = '1' else '0';
    o9  <= '1' after 0.3 ns when i = "01001" and
e'event and e = '1' else '0';
    o10 <= '1' after 0.3 ns when i = "01010" and
e'event and e = '1' else '0';
    o11 <= '1' after 0.3 ns when i = "01011" and
e'event and e = '1' else '0';
    o12 <= '1' after 0.3 ns when i = "01100" and
e'event and e = '1' else '0';
    o13 <= '1' after 0.3 ns when i = "01101" and
e'event and e = '1' else '0';
    o14 <= '1' after 0.3 ns when i = "01110" and
e'event and e = '1' else '0';
    o15 <= '1' after 0.3 ns when i = "01111" and
e'event and e = '1' else '0';
    o16 <= '1' after 0.3 ns when i = "10000" and
e'event and e = '1' else '0';
    o17 <= '1' after 0.3 ns when i = "10001" and
e'event and e = '1' else '0';
    o18 <= '1' after 0.3 ns when i = "10010" and
e'event and e = '1' else '0';
    o19 <= '1' after 0.3 ns when i = "10011" and
e'event and e = '1' else '0';
    o20 <= '1' after 0.3 ns when i = "10100" and
e'event and e = '1' else '0';
    o21 <= '1' after 0.3 ns when i = "10101" and
e'event and e = '1' else '0';
    o22 <= '1' after 0.3 ns when i = "10110" and
e'event and e = '1' else '0';
    o23 <= '1' after 0.3 ns when i = "10111" and
e'event and e = '1' else '0';
    o24 <= '1' after 0.3 ns when i = "11000" and
e'event and e = '1' else '0';
    o25 <= '1' after 0.3 ns when i = "11001" and
e'event and e = '1' else '0';
    o26 <= '1' after 0.3 ns when i = "11010" and
e'event and e = '1' else '0';
    o27 <= '1' after 0.3 ns when i = "11011" and
e'event and e = '1' else '0';
    o28 <= '1' after 0.3 ns when i = "11100" and
e'event and e = '1' else '0';
    o29 <= '1' after 0.3 ns when i = "11101" and
e'event and e = '1' else '0';
    o30 <= '1' after 0.3 ns when i = "11110" and
e'event and e = '1' else '0';
    o31 <= '1' after 0.3 ns when i = "11111" and
e'event and e = '1' else '0';
```

```
end stdece32_a;
```

STREG32

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_1164_extensions.all;

library my_packages;
use my_packages.stpack.all;

library st_comp;
use st_comp.streg8.all;

entity streg32 is
    port(data:   in  slv_32;
         ld:     in  std_logic;
         as:     in  std_logic;
         bs:     in  std_logic;
         vbt:    in  slv_4;
         db_set: in  std_logic;
         a:      out slv_32;
         b:      out slv_32;
         db:     out std_logic;
         p:      out slv_32);
end streg32;


architecture streg32_a of streg32 is

    component streg8
        port(i:  in  slv_8;
             ld: in  std_logic;
             as: in  std_logic;
             bs: in  std_logic;
             a:  out slv_8;
             b:  out slv_8;
             p:  out slv_8);
    end component;

    signal d0, d1, d2, d3: slv_8;
    signal ld0, ld1, ld2, ld3: std_logic;
    signal a0, a1, a2, a3: slv_8;
    signal b0, b1, b2, b3: slv_8;
    signal dbid, dbwb: std_logic := '0';
    signal p0, p1, p2, p3: slv_8;

begin

    d0 <= data(7 downto 0);
    d1 <= data(15 downto 8);
    d2 <= data(23 downto 16);
    d3 <= data(31 downto 24);

    ld0 <= vbt(0) and ld after 1 ns;
    ld1 <= vbt(1) and ld after 1 ns;
    ld2 <= vbt(2) and ld after 1 ns;
    ld3 <= vbt(3) and ld after 1 ns;

    reg_byte0: streg8
        port map(d0, ld0, as, bs, a0, b0, p0);

    reg_byte1: streg8
        port map(d1, ld1, as, bs, a1, b1, p1);

    reg_byte2: streg8
        port map(d2, ld2, as, bs, a2, b2, p2);

    reg_byte3: streg8
        port map(d3, ld3, as, bs, a3, b3, p3);

    a <= a3 & a2 & a1 & a0;
    b <= b3 & b2 & b1 & b0;

    p <= p3 & p2 & p1 & p0;

    dbid <= dbid xor '1' after 1 ns
        when db_set'event and db_set = '1'
        else dbid;

    dbwb <= dbwb xor '1' after 1 ns
        when ld'event and ld = '1'
        else dbwb;

    db <= dbid xor dbwb after 1 ns;

end streg32_a;
```

STREG8

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_1164_extensions.all;

library my_packages;
use my_packages.stpaok.all;

library st_comp;
use st_comp.sttg8.all;

entity streg8 is
    port(i:  in  slv_8;
         ld: in  std_logic;
         as: in  std_logic;
         bs: in  std_logic;
         a:  out slv_8;
         b:  out slv_8;
         p:  out slv_8);
end streg8;


architecture streg8_a of streg8 is

    component sttg8
        port(i:  in  slv_8;
             en: in  std_logic;
             o:  out slv_8);
    end component;

    signal ld_bar: std_logic := '0';
    signal m1: slv_8 := "00000000";
    signal m2: slv_8 := "00000000";
    signal m3: slv_8 := "00000000";

begin

    ld_bar <= not ld;

    input_tg:sttg8
        port map(i, ld, m1);

    m2 <= not m1 after 0.3 ns;

    m3 <= not m2 after 0.3 ns;

    feedback_tg:sttg8
        port map(m3, ld_bar, m1);

    a_bus_tg:sttg8
        port map(m3, as, a);

    b_bus_tg:sttg8
        port map(m3, bs, b);

    p <= m3;

end streg8_a;
```

STREGBANK

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_1164_extensions.all;

library my_packages;
use my_packages.stpack.all;

library st_comp;
use st_comp.streg32.all;
use st_comp.stdec32.all;
use st_comp.stdece32.all;

entity stregbank is
    port(data:     in  slv_32;
         reg_sel:  in  slv_5;
         a_sel:    in  slv_5;
         b_sel:    in  slv_5;
         db_sel:   in  slv_5;
         write:    in  std_logic;
         start:    in  std_logic;
         vbt:      in  slv_4;
         a:        out slv_32;
         b:        out slv_32;
         db:       out slv_32;
         r1_test:  out slv_32;
         r2_test:  out slv_32;
         r3_test:  out slv_32;
         r4_test:  out slv_32;
         r31_test: out slv_32);
```

```
end stregbank;


architecture stregbank_a of stregbank is

    component streg32
        port(data:   in  slv_32;
             ld:     in  std_logic;
             as:     in  std_logic;
             bs:     in  std_logic;
             vbt:    in  slv_4;
             db_set: in  std_logic;
             a:      out slv_32;
             b:      out slv_32;
             db:     out std_logic;
             p:      out slv_32);
    end component;

    component stdec32
        port(i:   in  slv_5;
             o0:  out std_logic;
             o1:  out std_logic;
             o2:  out std_logic;
             o3:  out std_logic;
             o4:  out std_logic;
             o5:  out std_logic;
             o6:  out std_logic;
             o7:  out std_logic;
             o8:  out std_logic;
             o9:  out std_logic;
             o10: out std_logic;
             o11: out std_logic;
             o12: out std_logic;
             o13: out std_logic;
             o14: out std_logic;
             o15: out std_logic;
             o16: out std_logic;
             o17: out std_logic;
             o18: out std_logic;
             o19: out std_logic;
             o20: out std_logic;
             o21: out std_logic;
             o22: out std_logic;
             o23: out std_logic;
             o24: out std_logic;
             o25: out std_logic;
             o26: out std_logic;
             o27: out std_logic;
             o28: out std_logic;
             o29: out std_logic;
             o30: out std_logic;
             o31: out std_logic);
    end component;

    component stdece32
        port(i:   in  slv_5;
             e:   in  std_logic;
             o0:  out std_logic;
             o1:  out std_logic;
             o2:  out std_logic;
             o3:  out std_logic;
             o4:  out std_logic;
             o5:  out std_logic;
             o6:  out std_logic;
             o7:  out std_logic;
             o8:  out std_logic;
             o9:  out std_logic;
             o10: out std_logic;
             o11: out std_logic;
             o12: out std_logic;
             o13: out std_logic;
             o14: out std_logic;
             o15: out std_logic;
             o16: out std_logic;
             o17: out std_logic;
             o18: out std_logic;
             o19: out std_logic;
             o20: out std_logic;
             o21: out std_logic;
             o22: out std_logic;
             o23: out std_logic;
             o24: out std_logic;
             o25: out std_logic;
             o26: out std_logic;
             o27: out std_logic;
             o28: out std_logic;
             o29: out std_logic;
             o30: out std_logic;
             o31: out std_logic);
    end component;

    constant zero32_constant: slv_32 :=
"00000000000000000000000000000000";

    signal zero32: slv_32 := zero32_constant;

    signal ld0, ld1, ld2, ld3, ld4, ld5, ld6, ld7:
std_logic;
    signal ld8, ld9, ld10, ld11, ld12, ld13, ld14,
ld15: std_logic;
    signal ld16, ld17, ld18, ld19, ld20, ld21, ld22,
ld23: std_logic;
    signal ld24, ld25, ld26, ld27, ld28, ld29, ld30,
ld31: std_logic;

    signal as0, as1, as2, as3, as4, as5, as6, as7:
std_logic;
    signal as8, as9, as10, as11, as12, as13, as14,
as15: std_logic;
    signal as16, as17, as18, as19, as20, as21, as22,
as23: std_logic;
    signal as24, as25, as26, as27, as28, as29, as30,
as31: std_logic;

    signal bs0, bs1, bs2, bs3, bs4, bs5, bs6, bs7:
std_logic;
    signal bs8, bs9, bs10, bs11, bs12, bs13, bs14,
bs15: std_logic;
    signal bs16, bs17, bs18, bs19, bs20, bs21, bs22,
bs23: std_logic;
    signal bs24, bs25, bs26, bs27, bs28, bs29, bs30,
bs31: std_logic;

    signal dbs0, dbs1, dbs2, dbs3, dbs4, dbs5, dbs6,
dbs7: std_logic;
    signal dbs8, dbs9, dbs10, dbs11, dbs12, dbs13,
dbs14, dbs15: std_logic;
    signal dbs16, dbs17, dbs18, dbs19, dbs20, dbs21,
dbs22, dbs23: std_logic;
    signal dbs24, dbs25, dbs26, dbs27, dbs28, dbs29,
dbs30, dbs31: std_logic;

    signal db0, db1, db2, db3, db4, db5, db6, db7:
std_logic;
    signal db8, db9, db10, db11, db12, db13, db14,
db15: std_logic;
    signal db16, db17, db18, db19, db20, db21, db22,
db23: std_logic;
    signal db24, db25, db26, db27, db28, db29, db30,
db31: std_logic;

    signal p0, p5, p6, p7: slv_32;
    signal p8, p9, p10, p11, p12, p13, p14, p15:
slv_32;
    signal p16, p17, p18, p19, p20, p21, p22, p23:
slv_32;
    signal p24, p25, p26, p27, p28, p29, p30: slv_32;

begin

    reg_sel_dec: stdece32
        port map(reg_sel, write, ld0, ld1, ld2, ld3,
ld4, ld5, ld6, ld7,
                 ld8, ld9, ld10, ld11, ld12, ld13,
ld14, ld15,
                 ld16, ld17, ld18, ld19, ld20, ld21,
ld22, ld23,
                 ld24, ld25, ld26, ld27, ld28, ld29,
ld30, ld31);

    a_sel_dec: stdec32
        port map(a_sel, as0, as1, as2, as3, as4, as5,
as6, as7,
                 as8, as9, as10, as11, as12, as13,
as14, as15,
                 as16, as17, as18, as19, as20, as21,
as22, as23,
                 as24, as25, as26, as27, as28, as29,
as30, as31);

    b_sel_dec: stdec32
        port map(b_sel, bs0, bs1, bs2, bs3, bs4, bs5,
bs6, bs7,
                 bs8, bs9, bs10, bs11, bs12, bs13,
bs14, bs15,
                 bs16, bs17, bs18, bs19, bs20, bs21,
bs22, bs23,
                 bs24, bs25, bs26, bs27, bs28, bs29,
bs30, bs31);

    db_sel_dec: stdece32
        port map(db_sel, start, dbs0, dbs1, dbs2, dbs3,
dbs4, dbs5, dbs6, dbs7,
                 dbs8, dbs9, dbs10, dbs11, dbs12,
dbs13, dbs14, dbs15,
                 dbs16, dbs17, dbs18, dbs19, dbs20,
dbs21, dbs22, dbs23,
```

```
                dbs24, dbs25, dbs26, dbs27, dbs28,
dbs29, dbs30, dbs31);

    reg0: streg32
        port map(zero32, ld0, as0,
                 bs0, vbt, dbs0, a, b, db0, p0);

    reg1: streg32
        port map(data, ld1, as1, bs1, vbt, dbs1, a, b,
db1, r1_test);

    reg2: streg32
        port map(data, ld2, as2, bs2, vbt, dbs2, a, b,
db2, r2_test);

    reg3: streg32
        port map(data, ld3, as3, bs3, vbt, dbs3, a, b,
db3, r3_test);

    reg4: streg32
        port map(data, ld4, as4, bs4, vbt, dbs4, a, b,
db4, r4_test);

    reg5: streg32
        port map(data, ld5, as5, bs5, vbt, dbs5, a, b,
db5, p5);

    reg6: streg32
        port map(data, ld6, as6, bs6, vbt, dbs6, a, b,
db6, p6);

    reg7: streg32
        port map(data, ld7, as7, bs7, vbt, dbs7, a, b,
db7, p7);

    reg8: streg32
        port map(data, ld8, as8, bs8, vbt, dbs8, a, b,
db8, p8);

    reg9: streg32
        port map(data, ld9, as9, bs9, vbt, dbs9, a, b,
db9, p9);

    reg10: streg32
        port map(data, ld10, as10, bs10, vbt, dbs10, a,
b, db10, p10);

    reg11: streg32
        port map(data, ld11, as11, bs11, vbt, dbs11, a,
b, db11, p11);

    reg12: streg32
        port map(data, ld12, as12, bs12, vbt, dbs12, a,
b, db12, p12);

    reg13: streg32
        port map(data, ld13, as13, bs13, vbt, dbs13, a,
b, db13, p13);

    reg14: streg32
        port map(data, ld14, as14, bs14, vbt, dbs14, a,
b, db14, p14);

    reg15: streg32
        port map(data, ld15, as15, bs15, vbt, dbs15, a,
b, db15, p15);

    reg16: streg32
        port map(data, ld16, as16, bs16, vbt, dbs16, a,
b, db16, p16);

    reg17: streg32
        port map(data, ld17, as17, bs17, vbt, dbs17, a,
b, db17, p17);

    reg18: streg32
        port map(data, ld18, as18, bs18, vbt, dbs18, a,
b, db18, p18);

    reg19: streg32
        port map(data, ld19, as19, bs19, vbt, dbs19, a,
b, db19, p19);

    reg20: streg32
        port map(data, ld20, as20, bs20, vbt, dbs20, a,
b, db20, p20);

    reg21: streg32
        port map(data, ld21, as21, bs21, vbt, dbs21, a,
b, db21, p21);

    reg22: streg32
        port map(data, ld22, as22, bs22, vbt, dbs22, a,
b, db22, p22);

    reg23: streg32
        port map(data, ld23, as23, bs23, vbt, dbs23, a,
b, db23, p23);

    reg24: streg32
        port map(data, ld24, as24, bs24, vbt, dbs24, a,
b, db24, p24);

    reg25: streg32
        port map(data, ld25, as25, bs25, vbt, dbs25, a,
b, db25, p25);

    reg26: streg32
        port map(data, ld26, as26, bs26, vbt, dbs26, a,
b, db26, p26);

    reg27: streg32
        port map(data, ld27, as27, bs27, vbt, dbs27, a,
b, db27, p27);

    reg28: streg32
        port map(data, ld28, as28, bs28, vbt, dbs28, a,
b, db28, p28);

    reg29: streg32
        port map(data, ld29, as29, bs29, vbt, dbs29, a,
b, db29, p29);

    reg30: streg32
        port map(data, ld30, as30, bs30, vbt, dbs30, a,
b, db30, p30);

    reg31: streg32
        port map(data, ld31, as31, bs31, vbt, dbs31, a,
b, db31, r31_test);

    db <= db31 & db30 & db29 & db28 & db27 & db26 &
db25 & db24 &
          db23 & db22 & db21 & db20 & db19 & db18 &
db17 & db16 &
          db15 & db14 & db13 & db12 & db11 & db10 & db9
& db8 &
          db7 & db6 & db5 & db4 & db3 & db2 & db1 &
'0';

end streg8bank_a;
```

STTG3

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_1164_extensions.all;

library my_packages;
use my_packages.stpack.all;

entity sttg8 is
    port(i:  in  slv_8;
         en: in  std_logic;
         o:  out slv_8);
end sttg8;


architecture sttg8_a of sttg8 is
begin
    o <= i after 0.1 ns when en = '1' else
         "ZZZZZZZZ" after 0.1 ns;
end sttg8_a;
```

APPENDIX E - SUPPORT PROGRAMS

MASS.C - MIPS ASSEMBLER

```
#include <stdio.h>
#include <math.h>

#define WHITE_SPACE_CHAR  token == ' ' ||  \
                          token == '\n' || \
                          token == '\t'
#define COMMENT           token == '#'
#define NOT_NEWLINE       token != '\n'
#define DIRECTIVE         token == '.'
#define NOT_EOF           token != EOF
#define DATA              strcmp(directive, "data") ==
0
#define CODE              strcmp(directive, "code") ==
0
#define NOT_DIRECTIVE     token != '.'
#define END               strcmp(directive, "end") == 0
#define NOT_DELIMITER     token != ' ' &&  \
                          token != '\n' && \
                          token != '\t' && \
                          token != ','
#define DATA_DIRECTIVE    strcmp(tokens, ".data") == 0
#define END_DIRECTIVE     strcmp(tokens, ".end") == 0

#define PAD5 "00000"
#define PAD10 "0000000000"
#define PAD15 "000000000000000"
#define PAD20 "00000000000000000000"
#define NEWLINE "\n"

#define NOP  "00000000000000000000000000000000"
#define HALT "11111100000000000000000000000000"

#define SPECIAL "000000"

#define SLL     "000000"
#define SRL     "000010"
#define SRA     "000011"
#define SLLV    "000100"
#define SRLV    "000110"
#define SRAV    "000111"

#define JR      "001000"
#define JALR    "001001"
#define SYSCALL "001100"
#define BREAK   "001101"

#define MFHI    "010000"
#define MTHI    "010001"
#define MFLO    "010010"
#define MTLO    "010011"

#define MULT    "011000"
#define MULTU   "011001"
#define DIV     "011010"
#define DIVU    "011011"

#define ADD     "100000"
#define ADDU    "100001"
#define SUB     "100010"
#define SUBU    "100011"
#define AND     "100100"
#define OR      "100101"
#define XOR     "100110"
#define NOR     "100111"

#define SLT     "101010"
#define SLTU    "101011"


#define BCOND   "000001"

#define BLTZ    "00000"
#define BGEZ    "00001"
#define BLTZAL  "10000"
#define BGEZAL  "10001"


#define J       "000010"
#define JAL     "000011"
#define BEQ     "000100"
#define BNE     "000101"
#define BLEZ    "000110"
#define BGTZ    "000111"

#define ADDI    "001000"
#define ADDIU   "001001"
#define SLTI    "001010"
#define SLTIU   "001011"
#define ANDI    "001100"
#define ORI     "001101"
#define XORI    "001110"
#define LUI     "001111"

#define LB      "100000"
#define LH      "100001"
#define LWL     "100010"
#define LW      "100011"
#define LBU     "100100"
#define LHU     "100101"
#define LWR     "100110"

#define SB      "101000"
#define SH      "101001"
#define SWL     "101010"
#define SW      "101011"
#define SWR     "101110"


enum boolean {FALSE, TRUE};
typedef enum boolean boolean;

FILE  * in;
FILE  * out;


void
itob5(int intval, char * binary)
{
    int i;

    for (i = 0; i <= 4; i++)
    {
        if (intval >= pow(2,(4-i)))
        {
            binary[i] = '1';
            intval = intval - pow(2,(4-i));
        }
        else
        {
            binary[i] = '0';
        }
    }
    binary[5] = '\0';
}


void
itob16(int intval, char * binary)
{
    int i;

    for (i = 0; i <= 15; i++)
    {
        if (intval >= pow(2,(15-i)))
        {
            binary[i] = '1';
            intval = intval - pow(2,(15-i));
        }
        else
        {
            binary[i] = '0';
        }
    }
    binary[16] = '\0';
}


void
itob26(int intval, char * binary)
{
    int i;

    for (i = 0; i <= 25; i++)
    {
        if (intval >= pow(2,(25-i)))
        {
            binary[i] = '1';
```

```
            intval = intval - pow(2,(25-i));
        }
        else
        {
            binary[i] = '0';
        }
    }
    binary[26] = '\0';
}


void
itob32(int intval, char * binary)
{
    int i;

    for (i = 0; i <= 31; i++)
    {
        if (intval >= pow(2,(31-i)))
        {
            binary[i] = '1';
            intval = intval - pow(2,(31-i));
        }
        else
        {
            binary[i] = '0';
        }
    }
    binary[32] = '\0';
}


void
outf(char * arg)
{
     fprintf(out, "%s", arg);
}


void
in5(char * binary)
{
  /*  char tokens[10];
    int arg;

    get_token(tokens);
    printf("%s ", tokens);
    if (tokens[0] == '$')
    {
        sscanf(tokens + 1, "%d", &arg);
    }
    itob5(arg, binary);*/

    int arg;

    fscanf(in, "%d", &arg);
    printf("%d ", arg);
    itob5(arg, binary);

}


void
in16(char * binary)
{
    int arg;

    fscanf(in, "%x", &arg);
    printf("%x ", arg);
    itob16(arg, binary);
}


void
in26(char * binary)
{
    int arg;

    fscanf(in, "%x", &arg);
    printf("%x ", arg);
    itob26(arg, binary);
}


void
message(void)
{
    printf("\n\n");
    printf(" ## MIPS ASSEMBLER, VERSION 1.0   ##\n");
    printf(" ## WRITTEN BY PAUL FANELLI, 1993 ##\n");
    printf("\n\n");
}
```

```
void
get_filename(char * name, char * mname)
{
    char temp[20];

    printf("ENTER FILENAME <== ");
    scanf("%s", temp);
    printf("\n\n");
    sprintf(name, "%s.test", temp);
    sprintf(mname, "%s.m", temp);
}


int
parse_data_seg_p1(void)
{
    char tokens[40];
    boolean done = FALSE;
    int data_count = 0;
    int eof = 0;

    printf("parse data segment\n");
    while ( done == FALSE && eof != EOF)
    {
        eof = get_token(tokens);
        if (END_DIRECTIVE)
        {
            done = TRUE;
        }
        else
        {
            data_count++;
        }
    }
    printf("%d data lines found\n", data_count);
    printf("end parse data segment\n");
    return data_count;
}


int
get_token(char * tokens)
{
    char token = ' ';
    int index = 0;
    boolean done = FALSE;

    while (NOT_EOF && done == FALSE)
    {
        token = fgetc(in);
        while (WHITE_SPACE_CHAR && NOT_EOF)
        {
            token = fgetc(in);
        }
        if (COMMENT)
        {
            while (NOT_NEWLINE)
            {
                token = fgetc(in);
            }
        }
        else
        {
            if (NOT_EOF)
            {
                while (NOT_DELIMITER && NOT_EOF)
                {
                    tokens[index] = token;
                    token = fgetc(in);
                    index++;
                }
                tokens[index] = '\0';
                done = TRUE;
            }
        }
    }
   /* printf("t = %s\n", tokens);*/
    return (int) token;
}


int
pass_one(void)
{
    char tokens[40];

    char directive[10];
    int data_count = 0;
    boolean done = FALSE;
    int eof = 0;

    printf("PASS ONE...\n");
```

```
    while(done == FALSE && eof != EOF)
    {
        eof = get_token(tokens);
        if (DATA_DIRECTIVE)
        {
            printf("found data segment\n");
            data_count = parse_data_seg_p1();
            printf("%d\n", data_count);
            done = TRUE;
        }
    }
    if (data_count == 0)
    {
        printf("no data segment found\n");
    }
    return data_count;
}


void
parse_data_seg_p2()
{
    char tokens[40];
    boolean done = FALSE;
    int  dat;
    char binary[33];
    int eof = 0;

    printf("parse data segment\n");
    while (done == FALSE && eof != EOF)
    {
        eof = get_token(tokens);
        if (END_DIRECTIVE)
        {
            done = TRUE;
        }
        else
        {
            sscanf(tokens, "%x", &dat);
            itob32(dat, binary);
            outf(binary);
            outf(NEWLINE);
            printf("%x, %d, %s\n", tokens, dat,
binary);
        }
    }
    printf("end parse data segment\n");
}


void
code(void)
{
    char  op[7];
    char  rs[6];
    char  rt[6];
    char  rd[6];
    char  shamt[6];
    char  base[6];
    char  offset[17];
    char  immed[17];
    char  target[27];
    char  token;


    while (get_token(op) != EOF)
    {
        printf("%s ", op);

        /* nop */
        if (strcmp(op, "nop") == 0)
        {
            outf(NOP);
            outf(NEWLINE);
        }

        /* sll rd, rt, shamt */
        else if (strcmp(op, "sll") == 0)
        {
            outf(SPECIAL);              /* opcode special
*/
            outf(PAD5);                 /* pad */
            in5(rd);                    /* get fields
*/
            in5(rt);
            in5(shamt);
            outf(rt);                   /* write fields
to file */
            outf(rd);
            outf(shamt);
            outf(SLL);                  /* function */
            outf(NEWLINE);
        }

        /* srl rd, rt, shamt */
        else if (strcmp(op, "srl") == 0)
        {
            outf(SPECIAL);              /* opcode special
*/
            outf(PAD5);                 /* pad */
            in5(rd);                    /* get fields
*/
            in5(rt);
            in5(shamt);
            outf(rt);                   /* write fields
to file */
            outf(rd);
            outf(shamt);
            outf(SRL);                  /* function */
            outf(NEWLINE);
        }

        /* sra rd, rt, shamt */
        else if (strcmp(op, "sra") == 0)
        {
            outf(SPECIAL);              /* opcode special
*/
            outf(PAD5);                 /* pad */
            in5(rd);                    /* get fields
*/
            in5(rt);
            in5(shamt);
            outf(rt);                   /* write fields
to file */
            outf(rd);
            outf(shamt);
            outf(SRA);                  /* function */
            outf(NEWLINE);
        }

        /* sllv rd, rt, rs */
        else if (strcmp(op, "sllv") == 0)
        {
            outf(SPECIAL);              /* opcode special
*/
            in5(rd);                    /* get fields
*/
            in5(rt);
            in5(rs);
            outf(rs);                   /* write fields
to file */
            outf(rt);
            outf(rd);
            outf(PAD5);                 /* pad */
            outf(SLLV);                 /* function */
            outf(NEWLINE);
        }

        /* srlv rd, rt, rs */
        else if (strcmp(op, "srlv") == 0)
        {
            outf(SPECIAL);              /* opcode special
*/
            in5(rd);                    /* get fields
*/
            in5(rt);
            in5(rs);
            outf(rs);                   /* write fields
to file */
            outf(rt);
            outf(rd);
            outf(PAD5);                 /* pad */
            outf(SRLV);                 /* function */
            outf(NEWLINE);
        }

        /* srav rd, rt, rs */
        else if (strcmp(op, "srav") == 0)
        {
            outf(SPECIAL);              /* opcode special
*/
            in5(rd);                    /* get fields
*/
            in5(rt);
            in5(rs);
            outf(rs);                   /* write fields
to file */
            outf(rt);
            outf(rd);
            outf(PAD5);                 /* pad */
            outf(SRAV);                 /* function */
            outf(NEWLINE);
        }

        /* jr rs */
        else if (strcmp(op, "jr") == 0)
```

```
        {
            outf(SPECIAL);              /* opcode special
*/
            in5(rs);                    /* get fields
*/
            outf(rs);                   /* write fields
to file */
            outf(PAD15);                /* pad */
            outf(JR);                   /* function */
            outf(NEWLINE);
        }

        /* jalr rd, rs */
        else if (strcmp(op, "jalr") == 0)
        {
            outf(SPECIAL);              /* opcode special
*/
            in5(rd);                    /* get fields
*/
            in5(rs);
            outf(rs);                   /* write fields
to file */
            outf(PAD5);                 /* pad */
            outf(rd);
            outf(PAD5);
            outf(JALR);                 /* function */
            outf(NEWLINE);
        }

        else if (strcmp(op, "syscall") == 0)
        {
            outf(SPECIAL);
            outf(PAD20);
            outf(SYSCALL);
            outf(NEWLINE);
        }

        else if (strcmp(op, "break") == 0)
        {
            outf(SPECIAL);
            outf(PAD20);        /* ??? */
            outf(BREAK);
            outf(NEWLINE);
        }

        /* mfhi rd */
        else if (strcmp(op, "mfhi") == 0)
        {
            outf("000000");
            outf(PAD10);
            in5(rd);
            outf(rd);
            outf(PAD5);
            outf(MFHI);
            outf(NEWLINE);
        }

        /* mthi rs */
        else if (strcmp(op, "mthi") == 0)
        {
            outf(SPECIAL);
            in5(rs);
            outf(rs);
            outf(PAD15);
            outf(MTHI);
            outf(NEWLINE);
        }

        /* mflo rd */
        else if (strcmp(op, "mflo") == 0)
        {
            outf(SPECIAL);
            outf(PAD10);
            in5(rd);
            outf(rd);
            outf(PAD5);
            outf(MFLO);
            outf(NEWLINE);
        }

        /* mtlo rs */
        else if (strcmp(op, "mtlo") == 0)
        {
            outf(SPECIAL);
            in5(rs);
            outf(rs);
            outf(PAD15);
            outf(MTLO);
            outf(NEWLINE);
        }

        /* mult rs, rt */
        else if (strcmp(op, "mult") == 0)
```

```
        {
            outf(SPECIAL);
            in5(rs);
            in5(rt);
            outf(rs);
            outf(rt);
            outf(PAD10);
            outf(MULT);
            outf(NEWLINE);
        }

        /* multu rs, rt */
        else if (strcmp(op, "multu") == 0)
        {
            outf(SPECIAL);
            in5(rs);
            in5(rt);
            outf(rs);
            outf(rt);
            outf(PAD10);
            outf(MULTU);
            outf(NEWLINE);
        }

        /* div rs, rt */
        else if (strcmp(op, "div") == 0)
        {
            outf(SPECIAL);
            in5(rs);
            in5(rt);
            outf(rs);
            outf(rt);
            outf(PAD10);
            outf(DIV);
            outf(NEWLINE);
        }

        /* divu rs, rt */
        else if (strcmp(op, "divu") == 0)
        {
            outf(SPECIAL);
            in5(rs);
            in5(rt);
            outf(rs);
            outf(rt);
            outf(PAD10);
            outf(DIVU);
            outf(NEWLINE);
        }

        /* add rd, rs, rt */
        else if (strcmp(op, "add") == 0)
        {
            outf(SPECIAL);
            in5(rd);
            in5(rs);
            in5(rt);
            outf(rs);
            outf(rt);
            outf(rd);
            outf(PAD5);
            outf(ADD);
            outf(NEWLINE);
        }

        /* addu rd, rs, rt */
        else if (strcmp(op, "addu") == 0)
        {
            outf(SPECIAL);
            in5(rd);
            in5(rs);
            in5(rt);
            outf(rs);
            outf(rt);
            outf(rd);
            outf(PAD5);
            outf(ADDU);
            outf(NEWLINE);
        }

        /* sub rd, rs, rt */
        else if (strcmp(op, "sub") == 0)
        {
            outf(SPECIAL);
            in5(rd);
            in5(rs);
            in5(rt);
            outf(rs);
            outf(rt);
            outf(rd);
            outf(PAD5);
            outf(SUB);
            outf(NEWLINE);
```

```
}

/* subu rd, rs, rt */
else if (strcmp(op, "subu") == 0)
{
    outf(SPECIAL);
    in5(rd);
    in5(rs);
    in5(rt);
    outf(rs);
    outf(rt);
    outf(rd);
    outf(PAD5);
    outf(SUBU);
    outf(NEWLINE);
}

/* and rd, rs, rt */
else if (strcmp(op, "and") == 0)
{
    outf(SPECIAL);
    in5(rd);
    in5(rs);
    in5(rt);
    outf(rs);
    outf(rt);
    outf(rd);
    outf(PAD5);
    outf(AND);
    outf(NEWLINE);
}

/* or rd, rs, rt */
else if (strcmp(op, "or") == 0)
{
    outf(SPECIAL);
    in5(rd);
    in5(rs);
    in5(rt);
    outf(rs);
    outf(rt);
    outf(rd);
    outf(PAD5);
    outf(OR);
    outf(NEWLINE);
}

/* xor rd, rs, rt */
else if (strcmp(op, "xor") == 0)
{
    outf(SPECIAL);
    in5(rd);
    in5(rs);
    in5(rt);
    outf(rs);
    outf(rt);
    outf(rd);
    outf(PAD5);
    outf(XOR);
    outf(NEWLINE);
}

/* nor rd, rs, rt */
else if (strcmp(op, "nor") == 0)
{
    outf(SPECIAL);
    in5(rd);
    in5(rs);
    in5(rt);
    outf(rs);
    outf(rt);
    outf(rd);
    outf(PAD5);
    outf(NOR);
    outf(NEWLINE);
}

/* slt rd, rs, rt */
else if (strcmp(op, "slt") == 0)
{
    outf(SPECIAL);
    in5(rd);
    in5(rs);
    in5(rt);
    outf(rs);
    outf(rt);
    outf(rd);
    outf(PAD5);
    outf(SLT);
    outf(NEWLINE);
}

/* sltu rd, rs, rt */
else if (strcmp(op, "sltu") == 0)
{
    outf(SPECIAL);
    in5(rd);
    in5(rs);
    in5(rt);
    outf(rs);
    outf(rt);
    outf(rd);
    outf(PAD5);
    outf(SLTU);
    outf(NEWLINE);
}

/* bltz rs, offset */
else if (strcmp(op, "bltz") == 0)
{
    outf(BCOND);
    in5(rs);
    outf(rs);
    outf(BLTZ);
    in16(offset);
    outf(offset);
    outf(NEWLINE);
}

/* bgez rs, offset */
else if (strcmp(op, "bgez") == 0)
{
    outf(BCOND);
    in5(rs);
    outf(rs);
    outf(BGEZ);
    in16(offset);
    outf(offset);
    outf(NEWLINE);
}

/* bltzal rs, offset */
else if (strcmp(op, "bltzal") == 0)
{
    outf(BCOND);
    in5(rs);
    outf(rs);
    outf(BLTZAL);
    in16(offset);
    outf(offset);
    outf(NEWLINE);
}

/* bgezal rs, offset */
else if (strcmp(op, "bgezal") == 0)
{
    outf(BCOND);
    in5(rs);
    outf(rs);
    outf(BGEZAL);
    in16(offset);
    outf(offset);
    outf(NEWLINE);
}

/* j target */
else if (strcmp(op, "j") == 0)
{
    outf(J);
    in26(target);
    outf(target);
    outf(NEWLINE);
}

/* jal target */
else if (strcmp(op, "jal") == 0)
{
    outf(JAL);
    in26(target);
    outf(target);
    outf(NEWLINE);
}

/* beq rs, rt, offset */
else if (strcmp(op, "beq") == 0)
{
    outf(BEQ);
    in5(rs);
    in5(rt);
    in16(offset);
    outf(rs);
    outf(rt);
    outf(offset);
    outf(NEWLINE);
}
```

```
/* bne rs, rt, offset */
else if (strcmp(op, "bne") == 0)
{
    outf(BNE);
    in5(rs);
    in5(rt);
    in16(offset);
    outf(rs);
    outf(rt);
    outf(offset);
    outf(NEWLINE);
}

/* blez rs, offset */
else if (strcmp(op, "blez") == 0)
{
    outf(BLEZ);
    in5(rs);
    in16(offset);
    outf(rs);
    outf(PAD5);
    outf(offset);
    outf(NEWLINE);
}

/* bgtz rs, offset */
else if (strcmp(op, "bgtz") == 0)
{
    outf(BGTZ);
    in5(rs);
    in16(offset);
    outf(rs);
    outf(PAD5);
    outf(offset);
    outf(NEWLINE);
}

/* addi rt, rs, immediate */
else if (strcmp(op, "addi") == 0)
{
    outf(ADDI);
    in5(rt);
    in5(rs);
    in16(immed);
    outf(rs);
    outf(rt);
    outf(immed);
    outf(NEWLINE);
}

/* addiu rt, rs, immediate */
else if (strcmp(op, "addiu") == 0)
{
    outf(ADDIU);
    in5(rt);
    in5(rs);
    in16(immed);
    outf(rs);
    outf(rt);
    outf(immed);
    outf(NEWLINE);
}

/* slti rt, rs, immediate */
else if (strcmp(op, "slti") == 0)
{
    outf(SLTI);
    in5(rt);
    in5(rs);
    in16(immed);
    outf(rs);
    outf(rt);
    outf(immed);
    outf(NEWLINE);
}

/* sltiu rt, rs, immediate */
else if (strcmp(op, "sltiu") == 0)
{
    outf(SLTIU);
    in5(rt);
    in5(rs);
    in16(immed);
    outf(rs);
    outf(rt);
    outf(immed);
    outf(NEWLINE);
}

/* andi rt, rs, immediate */
else if (strcmp(op, "andi") == 0)
{
    outf(ANDI);
    in5(rt);
    in5(rs);
    in16(immed);
    outf(rs);
    outf(rt);
    outf(immed);
    outf(NEWLINE);
}

/* ori rt, rs, immediate */
else if (strcmp(op, "ori") == 0)
{
    outf(ORI);
    in5(rt);
    in5(rs);
    in16(immed);
    outf(rs);
    outf(rt);
    outf(immed);
    outf(NEWLINE);
}

/* xori rt, rs, immediate */
else if (strcmp(op, "xori") == 0)
{
    outf(XORI);
    in5(rt);
    in5(rs);
    in16(immed);
    outf(rs);
    outf(rt);
    outf(immed);
    outf(NEWLINE);
}

/* lui rt, immediate */
else if (strcmp(op, "lui") == 0)
{
    outf(LUI);
    in5(rt);
    in16(immed);
    outf(PAD5);
    outf(rt);
    outf(immed);
    outf(NEWLINE);
}

/* lb rt, offset(base) */
else if (strcmp(op, "lb") == 0)
{
    outf(LB);
    in5(rt);
    in16(offset);
    in5(base);
    outf(base);
    outf(rt);
    outf(offset);
    outf(NEWLINE);
}

/* lh rt, offset(base) */
else if (strcmp(op, "lh") == 0)
{
    outf(LH);
    in5(rt);
    in16(offset);
    in5(base);
    outf(base);
    outf(rt);
    outf(offset);
    outf(NEWLINE);
}

/* lwl rt, offset(base) */
else if (strcmp(op, "lwl") == 0)
{
    outf(LWL);
    in5(rt);
    in16(offset);
    in5(base);
    outf(base);
    outf(rt);
    outf(offset);
    outf(NEWLINE);
}

/* lw rt, offset(base) */
else if (strcmp(op, "lw") == 0)
{
    outf(LW);
    in5(rt);
    in16(offset);
    in5(base);
```

```
        outf(base);
        outf(rt);
        outf(offset);
        outf(NEWLINE);
    }

    /* lbu rt, offset(base) */
    else if (strcmp(op, "lbu") == 0)
    {
        outf(LBU);
        in5(rt);
        in16(offset);
        in5(base);
        outf(base);
        outf(rt);
        outf(offset);
        outf(NEWLINE);
    }

    /* lhu rt, offset(base) */
    else if (strcmp(op, "lhu") == 0)
    {
        outf(LHU);
        in5(rt);
        in16(offset);
        in5(base);
        outf(base);
        outf(rt);
        outf(offset);
        outf(NEWLINE);
    }

    /* lwr rt, offset(base) */
    else if (strcmp(op, "lwr") == 0)
    {
        outf(LWH);
        in5(rt);
        in16(offset);
        in5(base);
        outf(base);
        outf(rt);
        outf(offset);
        outf(NEWLINE);
    }

    /* sb rt, offset(base) */
    else if (strcmp(op, "sb") == 0)
    {
        outf(SB);
        in5(rt);
        in16(offset);
        in5(base);
        outf(base);
        outf(rt);
        outf(offset);
        outf(NEWLINE);
    }

    /* sh rt, offset(base) */
    else if (strcmp(op, "sh") == 0)
    {
        outf(SH);
        in5(rt);
        in16(offset);
        in5(base);
        outf(base);
        outf(rt);
        outf(offset);
        outf(NEWLINE);
    }

    /*swl rt, offset(base) */
    else if (strcmp(op, "swl") == 0)
    {
        outf(SWL);
        in5(rt);
        in16(offset);
        in5(base);
        outf(base);
        outf(rt);
        outf(offset);
        outf(NEWLINE);
    }

    /* sw rt, offset(base) */
    else if (strcmp(op, "sw") == 0)
    {
        outf(SW);
        in5(rt);
        in16(offset);
        in5(base);
        outf(base);
        outf(rt);
        outf(offset);
        outf(NEWLINE);
    }

    /*swr rt, offset(base) */
    else if (strcmp(op, "swr") == 0)
    {
        outf(SWR);
        in5(rt);
        in16(offset);
        in5(base);
        outf(base);
        outf(rt);
        outf(offset);
        outf(NEWLINE);
    }

    /* halt instruction */
    else if (strcmp(op, "halt") == 0)
    {
        outf(HALT);
        outf(NEWLINE);
    }

    else
    {
        printf("\nERROR: UNKNOWN COMMAND");
    }

    printf("\n");

  } /* END WHILE FSCANF != EOF */

} /* END CODE FUNCTION */


void
pass_two(int data_count)
{
    char binary[33];
    char tokens[40];
    boolean done = FALSE;
    int eof = 0;

    printf("PASS TWO...\n");
    if (data_count != 0)
    {
        /* ADD JUMP INSTRUCTION TO JUMP OVER DATA
SEGMENT */
        outf(J);
        itob26(data_count + 2, binary);
        outf(binary);
        outf(NEWLINE);
        /* ADD NOP TO FILL DELAY SLOT */
        outf(NOP);
        outf(NEWLINE);
        /* GET DATA SEGMENT */
        while (done == FALSE && eof != EOF)
        {
            eof = get_token(tokens);
            if (DATA_DIRECTIVE)
            {
                parse_data_seg_p2();
                done = TRUE;
            }
        }

    }
    code();
}


int
main(void)
{
    char name[20];
    char mname[20];
    int data_count;

    message();
    get_filename(name, mname);

    in = fopen(name, "r");
    data_count = pass_one();
    fclose(in);

    in = fopen(name, "r");
    out = fopen(mname, "w");
    pass_two(data_count);
    fclose(in);
    fclose(out);

    printf("CONVERSION DONE\n");
```

```
    return 0;
}
```

MERA.C - MIPS EXPECTED RESULTS ASSEMBLER

```
# include <stdio.h>
# include <math.h>

#define WHITE_SPACE_CHAR  token == ' ' ||  \
                          token == '\n' || \
                          token == '\t'
#define COMMENT           token == '#'
#define NOT_EOF           token != EOF
#define NOT_DELIMITER     token != '#' && token != '\n'
#define NOT_NEWLINE       token != '\n'

enum boolean {FALSE, TRUE};
typedef enum boolean boolean;

FILE  * in = NULL;
FILE  * out = NULL;

char assem[1000][80];
int expect[1000][10];
char modify[1000][10];
int num_of_inst = 0;
int index = 0;
boolean p_mode;


void
itob32(int integer, char * bits)
{
    int i;
    int temp;

    if (integer < 0)
    {
        temp = -(integer + 1);
    }
    else
    {
        temp = integer;
    }
    for (i = 0; i <= 31; i++)
    {
        if ((temp % 2) == 1)
        {
            bits[31-i] = '1';
        }
        else
        {
            bits[31-i] = '0';
        }
        temp = temp / 2;
    }
    if (integer < 0)
    {
        for (i = 0; i <= 31; i++)
        {
            if (bits[i] == '0')
            {
                bits[i] = '1';
            }
            else
            {
                bits[i] = '0';
            }
        }
    }
    bits[32] = '\0';
}


int
power2(int x)
{
    if (x == 0) return 1;
    else return 2 * power2(x-1);
}


int
btoi32(char * bits)
{
    int i;
    int result = 0;
    boolean negative;

    negative = (bits[0] == '1');

    for (i = 0; i <= 31; i++)
    {
        if (negative)
        {
            if (bits[i] == '0')
            {
                result = result - power2(31-i);
            }
        }
        else
        {
            if (bits[i] == '1')
            {
                result = result + power2(31-i);
            }
        }
    }

    if (negative)
    {
        return --result;
    }
    else
    {
        return result;
    }
}


void
message(void)
{
    printf("\n\n");
    printf("
################################################\n");
    printf(" ## MIPS EXPECTED RESULTS ASSEMBLER,
VERSION 1.1 ##\n");
    printf(" ##          WRITTEN BY PAUL FANELLI, 1993
##\n");
    printf("
################################################\n");
    printf("\n\n");
}


void
help_message(void)
{
    printf("===> USE THE (?) KEY FOR HELP AT THE MENU
PROMPTS\n");
    printf("===> USE THE (RETURN) KEY RETURN FROM
MENU\n");
    printf("\n\n");
}


void
command_help(void)
{
    printf("************************  COMMAND MENU
************************\n");
    printf("    (n)ew      - assemble new expected
file\n");
    printf("    (o)ld      - assemble old expected
file\n");
    printf("    (a)ssemble - return to assemble
menu\n");
    printf("    (c)lear    - clear all array values
(set to zero)\n");
    printf("    (v)iew     - view assembly code and
expected values\n");
    printf("    (s)ave     - save expected values to
expected file\n");
    printf("    (q)uit     - quit out of program\n");
    printf("    (?)help    - print out this help
listing\n");

printf("************************************************
*****************\n");
    printf("\n");
}


void
assemble_help(void)
{
    printf("************************  ASSEMBLE MENU
************************\n");
    printf("  STATE:\n");
    printf("    (p)c          - enter in value for pc
register in hex\n");
```

```
    printf("     r(1)          - enter in value for r1
register\n");
    printf("     r(2)          - enter in value for r2
register\n");
    printf("     r(3)          - enter in value for r3
register\n");
    printf("     r(4)          - enter in value for r4
register\n");
    printf("    (r)31          - enter in value for r31
(link) register\n");
    printf("    (h)i           - enter in value for hi
register\n");
    printf("    (l)o           - enter in value for lo
register\n");
    printf("    (e)pc          - enter in value for epc
register\n");
    printf("    (c)ause        - enter in value for
cause register\n");

    printf("  CONTROL:\n");
    printf("    (b)ackward     - go back one instruction
(propagation off)\n");
    printf("    (f)orward      - go forward one
instruction (propagation off)\n");
    printf("    (i)nstruction - goto instruction\n");
    printf("    (n)ext         - go on to next
instruction\n");
    printf("    (s)tate        - view present state\n");
    printf("    (v)iew         - view present
contents\n");
    printf("    (return)       - return from assemble
menu\n");
    printf("    (?)help        - print out this help
listing\n");

printf("************************************************
****************\n");
    printf("\n");
}


int
command_menu(void)
{
    int ch;
    printf("COMMAND(n,o,a,c,v,s,q,?) <== ");
    ch = getchar();
    if (ch == '\n')
    {
        ch = getchar();
    }
    printf("\n");
    return ch;
}


int
assemble_menu(void)
{
    int ch;
    printf("ASSEMBLE( STATE[p,1,2,3,4,r,h,l,e,c]
CONTROL[b,f,i,n,s,v,return,?] ) <== ");
    ch = getchar();
    if (ch == '\n')
    {
        ch = getchar();
    }
    printf("\n");
    return ch;
}


void
view(void)
{
    int i;

    printf("************************************
ASSEMBLY CODE and EXPECTED VALUES  *************");
    printf("************************\n\n");
    printf("  INST                                        PC
R1      R2      R3      R4      R31      HI      ");
    printf("LO         EPC       CAUSE\n");
    printf("-----------------------------------------
-----------------------------------------");
    printf("----------------------------\n");
    for (i = 0; i < num_of_inst; i++)
    {
        printf("# %3d =>  %-20s   ", i+1, assem[i]);
        printf("%8x%c %5d%c ", expect[i][0],
modify[i][0], expect[i][1], modify[i][1]);
        printf("%5d%c %5d%c ", expect[i][2],
modify[i][2], expect[i][3], modify[i][3]);
        printf("%5d%c %5d%c ", expect[i][4],
modify[i][4], expect[i][5], modify[i][5]);
        printf("%5d%c %5d%c  ", expect[i][6],
modify[i][6], expect[i][7], modify[i][7]);
        printf("%8x%c  %8x%c", expect[i][8],
modify[i][8], expect[i][9], modify[i][9]);
        printf("\n");
    }
    printf("\n");
}


void
view_state(int modify_flag)
{
    static char pc = ' ';
    static char r1 = ' ';
    static char r2 = ' ';
    static char r3 = ' ';
    static char r4 = ' ';
    static char r5 = ' ';
    static char hi = ' ';
    static char lo = ' ';
    static char epc = ' ';
    static char cause = ' ';
    int i;

    i = index;
    if (modify_flag == 0)
    {
        pc = '*';
    }
    else if (modify_flag == 1)
    {
        r1 = '*';
    }
    else if (modify_flag == 2)
    {
        r2 = '*';
    }
    else if (modify_flag == 3)
    {
        r3 = '*';
    }
    else if (modify_flag == 4)
    {
        r4 = '*';
    }
    else if (modify_flag == 5)
    {
        r5 = '*';
    }
    else if (modify_flag == 6)
    {
        hi = '*';
    }
    else if (modify_flag == 7)
    {
        lo = '*';
    }
    else if (modify_flag == 8)
    {
        epc = '*';
    }
    else if (modify_flag == 9)
    {
        cause = '*';
    }
    else if (modify_flag == 10)
    {
        pc = ' ';
        r1 = ' ';
        r2 = ' ';
        r3 = ' ';
        r4 = ' ';
        r5 = ' ';
        hi = ' ';
        lo = ' ';
        epc = ' ';
        cause = ' ';
    }
    else
    {
    }
    printf("  INST                                        PC%c
R1%c    R2%c    ", pc, r1, r2);
    printf("R3%c    R4%c    R31%c    HI%c    ", r3, r4,
r5, hi);
    printf("LO%c       EPC%c     CAUSE%c\n", lo, epc,
cause);
    printf("-----------------------------------------
-------------------------------");
    printf("-----------------------------------\n");
```

```
    printf("# %3d =>  %-20s   ", i+1, assem[i]);
    printf("%8x  %5d  %5d  %5d  ", expect[i][0],
expect[i][1], expect[i][2], expect[i][3]);
    printf("%5d  %5d  %5d  %5d   ", expect[i][4],
expect[i][5], expect[i][6], expect[i][7]);
    printf("%8x   %8x", expect[i][8], expect[i][9]);
    printf("\n\n");

}


int
load_assem_file(void)
{
    char token = ' ';
    char tokens[80];
    int i = 0;
    int j = 0;
    boolean done = FALSE;

    while (NOT_EOF)
    {
        while (NOT_EOF && done == FALSE)
        {
            token = fgetc(in);
            while (WHITE_SPACE_CHAR && NOT_EOF)
                token = fgetc(in);
            if (COMMENT)
            {
                while (NOT_NEWLINE)
                    token = fgetc(in);
            }
            else
            {
                if (NOT_EOF)
                {
                    while (NOT_NEWLINE && NOT_EOF)
                    {
                        tokens[j] = token;
                        token = fgetc(in);
                        if (COMMENT)
                            while (NOT_NEWLINE)
                                token = fgetc(in);
                        j++;
                    }
                    tokens[j] = '\0';
                    printf("inst = %s\n", tokens);
                    strcpy(assem[i], tokens);
                    done = TRUE;
                    j = 0;
                    i++;
                }
            }
        }
        done = FALSE;
    }
    printf("\n");
    return i;
}


void
load_expected_file(void)
{
    int j;
    int i;
    char binary[33];

    for (j = 0; j < num_of_inst; j++)
    {
        for (i = 0; i <= 9; i++)
        {
            fscanf(in, "%s", binary);
            expect[j][i] = btoi32(binary);
        }
    }
}


void
build_modify_array(void)
{
    int j;
    int i;

    for (j = 0; j < num_of_inst; j++)
    {
        for (i = 0; i <= 9; i++)
        {
            if (j == 0)
            {
                if (expect[0][i] == 0)
                {
                    modify[0][i] = ' ';
                }
                else
                {
                    modify[0][i] = '*';
                }
            }
            else
            {
                if (expect[j][i] == expect[j-1][i])
                {
                    modify[j][i] = ' ';
                }
                else
                {
                    modify[j][i] = '*';
                }
            }
        }
    }
}


void
initialize(void)
{
    int i;
    int j;

    for (j = 0; j <= 999; j++)
    {
        for (i = 0; i <= 9; i++)
        {
            expect[j][i] = 0;
            modify[j][i] = ' ';
        }
    }
}


boolean
get_filename(char * name, char * ename)
{
    char temp[20];
    int ch;

    printf("ENTER FILENAME (q to quit) <== ");
    scanf("%s", temp);
    printf("\n\n");
    if (strcmp(temp, "q") == 0)
        return TRUE;
    else
    {
        printf("IS THE TEST FILE FOR BRANCH/JUMP
(y,[n]) <== ");
        ch = getchar();
        if (ch == '\n')
        {
            ch = getchar();
        }
        printf("\n");
        if (ch == 'y')
        {
            sprintf(name, "%s.f", temp);
        }
        else
        {
            sprintf(name, "%s.test", temp);
        }
        sprintf(ename, "%s.e", temp);
        return FALSE;
    }
}



void
change_pc(void)
{
    char input[33];
    int temp;

    printf("OLD PC VALUE (HEX) = %x\n",
expect[index][0]);
    printf("NEW PC VALUE (HEX) - use (i) to increment
by 4 <== ");
    scanf("%s", input);
    printf("\n");
    if (input[0] == 'i')
    {
        expect[index][0] = expect[index][0] + 4;
    }
    else
```

```
    {
        sscanf(input, "%x", &temp);
        expect[index][0] = temp;
    }
    view_state(0);
}


int
get_input(void)
{
    char input[33];
    int temp;

    scanf("%s", input);
    printf("\n");
    if (input[0] == 'h')
    {
        sscanf(input+1, "%x", &temp);
    }
    else
    {
        sscanf(input, "%d", &temp);
    }
    return temp;
}


void
change_r1(void)
{
    printf("OLD R1 VALUE = %d\n", expect[index][1]);
    printf("NEW R1 VALUE - use (h) for hex <== ");
    expect[index][1] = get_input();
    view_state(1);
}


void
change_r2(void)
{
    int input;

    printf("OLD R2 VALUE = %d\n", expect[index][2]);
    printf("NEW R2 VALUE - use (h) for hex <== ");
    expect[index][2] = get_input();
    view_state(2);
}


void
change_r3(void)
{
    int input;

    printf("OLD R3 VALUE = %d\n", expect[index][3]);
    printf("NEW R3 VALUE - use (h) for hex <== ");
    expect[index][3] = get_input();
    view_state(3);
}


void
change_r4(void)
{
    int input;

    printf("OLD R4 VALUE = %d\n", expect[index][4]);
    printf("NEW R4 VALUE - use (h) for hex <== ");
    expect[index][4] = get_input();
    view_state(4);
}


void
change_r31(void)
{
    int input;

    printf("OLD R31 VALUE = %d\n", expect[index][5]);
    printf("NEW R31 VALUE - use (h) for hex <== ");
    expect[index][5] = get_input();
    view_state(5);
}


void
change_hi(void)
{
    int input;

    printf("OLD HI VALUE = %d\n", expect[index][6]);
    printf("NEW HI VALUE - use (h) for hex <== ");
    expect[index][6] = get_input();
    view_state(6);
}


void
change_lo(void)
{
    int input;

    printf("OLD LO VALUE = %d\n", expect[index][7]);
    printf("NEW LO VALUE - use (h) for hex <== ");
    expect[index][7] = get_input();
    view_state(7);
}


void
change_epc(void)
{
    int input;

    printf("OLD EPC VALUE (HEX) = %x\n",
expect[index][8]);
    printf("NEW EPC VALUE (HEX) <== ");
    scanf("%x", &input);
    printf("\n");
    expect[index][8] = input;
    view_state(8);
}


void
change_cause(void)
{
    int input;

    printf("OLD CAUSE VALUE (HEX) = %x\n",
expect[index][9]);
    printf("NEW CAUSE VALUE (HEX) <== ");
    scanf("%x", &input);
    printf("\n");
    expect[index][9] = input;
    view_state(9);
}


boolean
query_new_state(void)
{
    int selector;
    boolean done = FALSE;
    boolean next = FALSE;
    int i;

    while (done == FALSE && next == FALSE)
    {
        selector = assemble_menu();
        switch (selector)
        {
            case 'p':
                change_pc();
                break;
            case '1':
                change_r1();
                break;
            case '2':
                change_r2();
                break;
            case '3':
                change_r3();
                break;
            case '4':
                change_r4();
                break;
            case 'r':
                change_r31();
                break;
            case 'h':
                change_hi();
                break;
            case 'l':
                change_lo();
                break;
            case 'e':
                change_epc();
                break;
            case 'c':
                change_cause();
                break;
            case 'b':
                index = index - 1;
                next = TRUE;
```

```
                break;
            case 'f':
                index++;
                next = TRUE;
                break;
            case 'i':
                index = goto_instruction();
                if (index >= 0)
                    view_state(-1);
                break;
            case 'n':
                for (i = 0; i <= 9; i++)
                    expect[index+1][i] =
expect[index][i];
                index++;
                next = TRUE;
                break;
            case 's':
                view_state(-1);
                break;
            case 'v':
                view();
                break;
            case '?':
                assemble_help();
                break;
            case '\n':
                done = TRUE;
                break;
            default:
                printf("ILLEGAL COMMAND - TRY
AGAIN\n");
        }
    }
    return done;
}


void
modify_update(void)
{
    int i;

    for (i = 0; i <= 9; i++)
    {
        if (index == 0)
        {
            if (expect[0][i] == 0)
                modify[0][i] = ' ';
            else
                modify[0][i] = '*';
        }
        else
        {
            if (expect[index][i] == expect[index-1][i])
                modify[index][i] = ' ';
            else
                modify[index][i] = '*';
        }

        if (expect[index][i] == expect[index+1][i])
            modify[index+1][i] = ' ';
        else
            modify[index+1][i] = '*';
    }
}


void
assemble(void)
{
    int i;
    boolean done = FALSE;

    while (index < num_of_inst && done == FALSE)
    {
        view_state(10);
        done = query_new_state();
        modify_update();
    }
}


void
assemble_new_file(char * name, char * ename)
{
    if (get_filename(name, ename) == TRUE)
        return;
    in = fopen(name, "r");
    num_of_inst = load_assem_file();
    fclose(in);
    initialize();
    view();
    assemble();
}


int
goto_instruction(void)
{
    int input;

    printf("GOTO INSTRUCTION NUMBER - (0) for exit <==
");
    scanf("%d", &input);
    printf("\n");
    return (input - 1);
}


void
change_old_file(char * name, char * ename)
{
    if (get_filename(name, ename) == TRUE)
        return;
    in = fopen(name, "r");
    num_of_inst = load_assem_file();
    fclose(in);
    in = fopen(ename, "r");
    load_expected_file();
    fclose(in);
    build_modify_array();
    view();
    assemble();
}


void
save_file(char * ename)
{
    int j;
    int i;
    char binary[33];

    if (num_of_inst != 0)
    {
        out = fopen(ename, "w");
        for (j = 0; j < num_of_inst; j++)
        {
            for (i = 0; i <= 9; i++)
            {
                itob32(expect[j][i], binary);
                fprintf(out, "%s", binary);
                fprintf(out, "%s", "\n");
            }
        }
        fclose(out);
    }
    else
    {
        printf("!!! WARNING: NO DATA TO SAVE !!!\n");
    }
    printf("\n");
}


void
clear(void)
{
    int i,j;

    for (j = 0; j < 1000; j++)
        for (i = 0; i < 10; i++)
            expect[j][i] = 0;
}


boolean
quit()
{
    int ch;

    printf("DO YOU REALLY WANT TO QUIT (y,[n]) <== ");
    ch = getchar();
    if (ch == '\n')
    {
        ch = getchar();
    }
    printf("\n");
    if (ch == 'y')
    {
        return TRUE;
    }
    else
    {
        return FALSE;
```

```
    }
}


boolean
query_save(char * ename)
{
    int ch;

    printf("SAVE CHANGES TO %s (y,[n]) <== ", ename);
    ch = getchar();
    if (ch == '\n')
    {
        ch = getchar();
    }
    printf("\n");
    if (ch == 'y')
    {
        return TRUE;
    }
    else
    {
        return FALSE;
    }
}


int
main(void)
{
    int selector;
    boolean done = FALSE;
    char name[20];
    char ename[20];
    boolean save;

    message();
    help_message();
    while (done == FALSE)
    {
        selector = command_menu();
        switch (selector)
        {
            case 'n':
                index = 0;
                assemble_new_file(name, ename);
                save = TRUE;
                break;
            case 'o':
                index = 0;
                change_old_file(name, ename);
                save = TRUE;
                break;
            case 'a':
                assemble();
                save = TRUE;
                break;
            case 'c':
                clear();
                save = TRUE;
                break;
            case 'v':
                view();
                break;
            case 's':
                save_file(ename);
                save = FALSE;
                break;
            case 'q':
                if (save == TRUE)
                {
                    if (query_save(ename) == TRUE)
                    {
                        save_file(ename);
                        save = FALSE;
                    }
                }
                done = quit();
                break;
            case '?':
                command_help();
                break;
            default:
                printf("ILLEGAL COMMAND - TRY
AGAIN\n");
        }
    }
    return 0;
}
```

FLOW - PROGRAM TO UNWIND MIPS ASSEMBLY CODE WITH BRANCHES AND JUMPS

```
# include <stdio.h>

#define WHITE_SPACE_CHAR  token == ' ' ||  \
                          token == '\n' || \
                          token == '\t'
#define COMMENT           token == '#'
#define NOT_EOF           token != EOF
#define NOT_DELIMITER     token != '#' && token != '\n'
#define NOT_NEWLINE       token != '\n'

enum boolean {FALSE, TRUE};
typedef enum boolean boolean;

FILE  * in = NULL;
FILE  * out = NULL;
FILE  * out2 = NULL;

char assem[1000][80];
int order[1000];
int index = 0;
int start = 0;
int finish = 0;



void
message(void)
{
    printf("\n\n");
    printf("
##################################################\n");
    printf(" ##   MIPS PROGRAM FLOW ASSEMBLER, VERSION
1.1   ##\n");
    printf(" ##          WRITTEN BY PAUL FANELLI, 1993
##\n");
    printf("
##################################################\n");
    printf("\n\n");
}


void
help_message(void)
{
    printf("===> USE THE (?) KEY FOR HELP AT THE MENU
PROMPTS\n");
    printf("===> USE THE (RETURN) KEY RETURN FROM
MENU\n");
    printf("\n\n");
}


void
command_help(void)
{
    printf("*********************  COMMAND MENU
**********************\n");
    printf("    (n)ew       - assemble new expected
file\n");
    printf("    (o)ld       - assemble old expected
file\n");
    printf("    (a)ssemble - return to assemble
menu\n");
    printf("    (c)lear     - clear all array values
(set to zero)\n");
    printf("    (v)iew      - view assembly code and
expected values\n");
    printf("    (s)ave      - save expected values to
expected file\n");
    printf("    (q)uit      - quit out of program\n");
    printf("    (?)help     - print out this help
listing\n");

printf("***********************************************
*****************\n");
    printf("\n");
}


void
assemble_help(void)
{
    printf("*********************  ASSEMBLE MENU
**********************\n");
    printf("    (e)nter       - enter order value\n");
    printf("    (b)ackward    - go back one
instruction\n");
    printf("    (f)orward     - go forward one
instruction\n");
```

```
    printf("    (i)nstruction - goto instruction\n");
    printf("    (v)iew        - view present
contents\n");
    printf("    (s)tate       - view present state\n");
    printf("    (r)ange       - set view range\n");
    printf("    (return)      - return from assemble
menu\n");
    printf("    (?)help       - print out this help
listing\n");

printf("************************************************
*****************\n");
    printf("\n");
}


int
command_menu(void)
{
    int ch;
    printf("COMMAND(n,o,a,c,v,s,q,?) <== ");
    ch = getchar();
    if (ch == '\n')
    {
        ch = getchar();
    }
    printf("\n");
    return ch;
}


int
assemble_menu(void)
{
    int ch;
    printf("ASSEMBLE(e,b,f,i,v,s,r,return,?) <== ");
    ch = getchar();
    if (ch == '\n')
    {
        ch = getchar();
    }
    printf("\n");
    return ch;
}


void
view(void)
{
    int i;

    printf("********** ORIGINAL **********  **********
ORDERDED **********\n\n");
    for (i=start; i <= finish; i++)
        printf("# %3d => %-20s    %3d => %-20s\n", i,
assem[i], order[i], assem[order[i]]);
    printf("\n");
}


void
view_state(void)
{
    printf("INST # %3d, %3d => %-20s\n", index,
order[index], assem[order[index]]);
    printf("\n");
}


void
load_assem_file(void)
{
    char token = ' ';
    char tokens[80];
    int i = 0;
    int j = 0;
    boolean done = FALSE;

    while (NOT_EOF)
    {
        while (NOT_EOF && done == FALSE)
        {
            token = fgetc(in);
            while (WHITE_SPACE_CHAR && NOT_EOF)
                token = fgetc(in);
            if (COMMENT)
            {
                while (NOT_NEWLINE)
                    token = fgetc(in);
            }
            else
            {
                if (NOT_EOF)
                {
                    while (NOT_NEWLINE && NOT_EOF)
                    {
                        tokens[j] = token;
                        token = fgetc(in);
                        if (COMMENT)
                            while (NOT_NEWLINE)
                                token = fgetc(in);
                        j++;
                    }
                    tokens[j] = '\0';
                    printf("inst = %s\n", tokens);
                    strcpy(assem[i], tokens);
                    done = TRUE;
                    j = 0;
                    i++;
                }
            }
        }
        done = FALSE;
    }
    printf("\n");
    printf("NUMBER OF ASSEMBLY INSTRUCTIONS = %d\n",
i);
    printf("\n");
}


void
load_order_file(void)
{
    int i = 0;
    int integer;

    while (fscanf(in, "%d", &integer) != EOF)
    {
        order[i] = integer;
        i++;
    }
    printf("NUMBER OF ORDERED INSTRUCTIONS = %d\n", i);
    printf("\n");
}


void
initialize(void)
{
    int i;

    for (i = 0; i <= 999; i++)
    {
        order[i] = 999;
    }
}


boolean
get_filename(char * name, char * oname, char * fname)
{
    char temp[20];

    printf("ENTER FILENAME (q to quit) <== ");
    scanf("%s", temp);
    printf("\n\n");
    if (strcmp(temp, "q") == 0)
    {
        return TRUE;
    }
    else
    {
        sprintf(name, "%s.test", temp);
        sprintf(oname, "%s.o", temp);
        sprintf(fname, "%s.f", temp);
        return FALSE;
    }
}


int
get_input(void)
{
    int input;

    scanf("%d", &input);
    printf("\n");
    return input;
}


void
enter(void)
{
    printf("OLD VALUE = %d\n", order[index]);
```

```
    printf("NEW VALUE <== ");
    order[index] = get_input();
    printf("\n");
}


void
set_view_range(void)
{
    printf("ENTER VIEW RANGE (LO,HI) <== ");
    scanf("%d,%d", &start, &finish);
    printf("\n");
}


void
assemble(void)
{
    int selector;
    boolean done = FALSE;
    int i;

    while (done == FALSE)
    {
        view_state();
        selector = assemble_menu();
        switch (selector)
        {
            case 'e':
                enter();
                index++;
                break;
            case 'b':
                index--;
                break;
            case 'f':
                index++;
                break;
            case 'i':
                index = goto_instruction();
                break;
            case 'v':
                view();
                break;
            case 's':
                break;
            case 'r':
                set_view_range();
                break;
            case '?':
                assemble_help();
                break;
            case '\n':
                done = TRUE;
                break;
            default:
                printf("ILLEGAL COMMAND - TRY
AGAIN\n");
        }
    }
}



void
assemble_new_file(char * name, char * oname, char *
fname)
{
    if (get_filename(name, oname, fname) == TRUE)
        return;
    in = fopen(name, "r");
    load_assem_file();
    fclose(in);

    initialize();
    view();
    assemble();
}


int
goto_instruction(void)
{
    int input;

    printf("GOTO INSTRUCTION NUMBER <== ");
    scanf("%d", &input);
    printf("\n");
    return input;
}


void
change_old_file(char * name, char * oname, char *
fname)
{
    if (get_filename(name, oname, fname) == TRUE)
        return;
    in = fopen(name, "r");
    load_assem_file();
    fclose(in);
    in = fopen(oname, "r");
    initialize();
    load_order_file();
    fclose(in);
    view();
    assemble();
}


void
save_file(char * oname, char * fname)
{
    int j;

    if (index != 0)
    {
        out = fopen(fname, "w");
        out2 = fopen(oname, "w");
        for (j = 0; j < index; j++)
        {
            fprintf(out, "%s", assem[order[j]]);
            fprintf(out, "%s", "\n");
            fprintf(out2, "%d\n", order[j]);
        }
        fclose(out);
        fclose(out2);
    }
    else
    {
        printf("!!! WARNING: NO DATA TO SAVE !!!\n");
    }
    printf("\n");
}


void
clear(void)
{
    int i;

    for (i = 0; i < 1000; i++)
        order[i] = 999;
}


boolean
quit()
{
    int ch;

    printf("DO YOU REALLY WANT TO QUIT (y,[n]) <== ");
    ch = getchar();
    if (ch == '\n')
    {
        ch = getchar();
    }
    printf("\n");
    if (ch == 'y')
    {
        return TRUE;
    }
    else
    {
        return FALSE;
    }
}


boolean
query_save(char * oname)
{
    int ch;

    printf("SAVE CHANGES TO %s (y,[n]) <== ", oname);
    ch = getchar();
    if (ch == '\n')
    {
        ch = getchar();
    }
    printf("\n");
    if (ch == 'y')
    {
        return TRUE;
    }
    else
```

```
    {
        return FALSE;
    }
}


int
main(void)
{
    int selector;
    boolean done = FALSE;
    char name[20];
    char oname[20];
    char fname[20];
    boolean save;

    message();
    help_message();
    while (done == FALSE)
    {
        selector = command_menu();
        switch (selector)
        {
            case 'n':
                index = 0;
                assemble_new_file(name, oname, fname);
                save = TRUE;
                break;
            case 'o':
                index = 0;
                change_old_file(name, oname, fname);
                save = TRUE;
                break;
            case 'a':
                assemble();
                save = TRUE;
                break;
            case 'c':
                clear();
                save = TRUE;
                break;
            case 'v':
                view();
                break;
            case 's':
                save_file(oname, fname);
                save = FALSE;
                break;
            case 'q':
                if (save == TRUE)
                {
                    if (query_save(oname) == TRUE)
                    {
                        save_file(oname, fname);
                        save = FALSE;
                    }
                }
                done = quit();
                break;
            case '?':
                command_help();
                break;
            default:
                printf("ILLEGAL COMMAND - TRY
AGAIN\n");
        }
    }
    return 0;
}
```

MPP - MIPS PREPROCESSOR

```
# include <stdio.h>

enum boolean {FALSE, TRUE};
typedef enum boolean boolean;

FILE  * in = NULL;
FILE  * out = NULL;


void
get_filename(char * mname, char * ename)
{
    char temp[20];

    printf("ENTER FILENAME <== ");
    scanf("%s", temp);
    printf("\n\n");
    sprintf(mname, "%s.m", temp);
    sprintf(ename, "%s.e", temp);
}


void
copy_file(void)
{
    char token = ' ';

    while (token != EOF)
    {
        token = fgetc(in);
        if (token != EOF)
        {
            fputc(token, out);
        }
    }
}


int
main(void)
{
    char mname[20];
    char ename[20];

    printf("*** MIPS PREPROCESSOR ***\n\n");
    get_filename(mname, ename);

    in = fopen(mname, "r");
    out = fopen("machine", "w");
    copy_file();
    fclose(in);
    fclose(out);

    in = fopen(ename, "r");
    out = fopen("expected", "w");
    copy_file();
    fclose(in);
    fclose(out);

    printf("*** PREPROCESSOR  COMPLETE ***\n\n");

    return 0;
}
```

APPENDIX F - TEST PROGRAMS

ALTEST - ALU IMMEDIATE

```
# test file for alu immediate
arithmetic instructions

# lui - load upper immediate
lui 1 0
lui 1 1
lui 1 a
lui 1 f
lui 1 ffff

# ori - or immediate
ori 1 0 0
ori 2 1 5
ori 1 2 f
lui 1 5555
ori 1 1 5555
ori 2 1 0
ori 2 1 1
ori 2 1 2
ori 2 1 ffff

# andi - and immediate
andi 2 1 0
andi 2 1 1
andi 2 1 2
andi 2 1 ffff

# xori - exclusive or immediate
xori 2 1 0
xori 2 1 1
xori 2 1 2
xori 2 1 ffff

# addi - add immediate
ori 2 0 0
ori 1 0 0
addi 2 1 0
ori 1 0 a
addi 2 1 5
addi 2 1 fffb
lui 1 ffff
ori 1 1 fff6
addi 2 1 5
addi 2 1 fffb

# addiu - add immediate unsigned
ori 1 0 0
addiu 2 1 ffff
addiu 1 2 1
lui 1 7fff
ori 1 1 ffff
addiu 2 1 1
addiu 1 2 ffff

# slti - set on less than
immediate
ori 1 0 2
slti 2 1 ffff
slti 2 1 0
slti 2 1 1
slti 2 1 2
slti 2 1 3
addi 1 0 fffe
slti 2 1 fffd
slti 2 1 fffe
slti 2 1 ffff
slti 2 1 0
slti 2 1 1

# sltiu - set on less than
immediate unsigned
ori 1 0 2
sltiu 2 1 0
sltiu 2 1 1
sltiu 2 1 2
sltiu 2 1 3
halt
```

ARTEST - ALU REGISTER

```
# test file for arithmetic
instructions
# 3 operand, register type

# add - add
ori 1 0 0
ori 2 0 0
add 3 1 2

ori 1 0 a
ori 2 0 5
add 3 1 2
add 3 2 1

addi 2 0 fffb
add 3 1 2
add 3 2 1

addi 1 0 fff6
add 3 1 2
add 3 2 1

ori 2 0 5
add 3 1 2
add 3 2 1

ori 1 0 64
ori 2 0 c8
add 3 1 2

# addu - add unsigned
ori 1 0 0
ori 2 0 0
addu 3 1 2

ori 1 0 a
ori 2 0 5
addu 3 1 2
addu 3 2 1

addi 2 0 fffb
addu 3 1 2
addu 3 2 1

addi 1 0 fff6
addu 3 1 2
addu 3 2 1

ori 2 0 5
addu 3 1 2
addu 3 2 1

ori 1 0 64
ori 2 0 c8
addu 3 1 2

# sub - subtract
ori 1 0 0
ori 2 0 0
sub 3 1 2

ori 1 0 a
ori 2 0 5
sub 3 1 2
sub 3 2 1

addi 2 0 fffb
sub 3 1 2
sub 3 2 1

addi 1 0 fff6
sub 3 1 2
sub 3 2 1

ori 2 0 5
sub 3 1 2
sub 3 2 1

ori 1 0 64
ori 2 0 c8
sub 3 1 2
sub 3 2 1
```

```
# subu - subtract unsigned
ori 1 0 0
ori 2 0 0
subu 3 1 2

ori 1 0 a
ori 2 0 5
subu 3 1 2
subu 3 2 1

addi 2 0 fffb
subu 3 1 2
subu 3 2 1

addi 1 0 fff6
subu 3 1 2
subu 3 2 1

ori 2 0 5
subu 3 1 2
subu 3 2 1

ori 1 0 64
ori 2 0 c8
subu 3 1 2
subu 3 2 1

# slt - set on less than
ori 1 0 2
addi 2 0 ffff
slt 3 1 2
ori 2 0 0
slt 3 1 2
ori 2 0 1
slt 3 1 2
ori 2 0 2
slt 3 1 2
ori 2 0 3
slt 3 1 2

addi 1 0 fffe
addi 2 0 fffd
slt 3 1 2
addi 2 0 fffe
slt 3 1 2
addi 2 0 ffff
slt 3 1 2
ori 2 0 0
slt 3 1 2
ori 2 0 1
slt 3 1 2   # run for 6300

# sltu - set on less than
unsigned
ori 1 0 2
ori 2 0 0
sltu 3 1 2
ori 2 0 1
sltu 3 1 2
ori 2 0 2
sltu 3 1 2
ori 2 0 3
sltu 3 1 2   # run for 6900

# and - and
lui 1 5555
ori 1 1 5555
ori 2 0 0
and 3 1 2
ori 2 0 1
and 3 1 2
ori 2 0 2
and 3 1 2
ori 2 0 ffff
and 3 1 2

# or - or
ori 2 0 0
or 3 1 2
ori 2 0 1
or 3 1 2
ori 2 0 2
or 3 1 2
ori 2 0 ffff
```

```
or 3 1 2

# xor - exclusive or
ori 2 0 0
xor 3 1 2
ori 2 0 1
xor 3 1 2
ori 2 0 2
xor 3 1 2
ori 2 0 ffff
xor 3 1 2

# nor - nor
ori 2 0 0
nor 3 1 2
ori 2 0 1
nor 3 1 2
ori 2 0 2
nor 3 1 2
ori 2 0 ffff
nor 3 1 2    # run for 8900

halt
```

JB.TEST - JUMP AND BRANCH

```
# test file for jump and branch
instructions

# j - jump
j 8          #0h
ori 1 0 1
ori 1 0 2
nop
j 9          #10h
ori 1 0 4
nop
nop
j 4          #20h
ori 1 0 3

# jal - jump and link
jal 10
ori 1 0 5
ori 1 0 6   #30h
jal 11
nop
nop
jal c        #40h
ori 1 0 7

# jr - jump register
ori 2 0 60
jr 2
ori 1 0 8   #50h
ori 2 0 6c
jr 2
nop
ori 1 0 9   #60h
ori 2 0 54
jr 2
ori 1 0 a
nop          #70h

# jalr - jump and link register
ori 2 0 90
jalr 4 2
ori 1 0 b
nop          #80h
nop
nop
nop
ori 1 0 c   #90h

# beq - branch on equal
ori 2 0 abcd
ori 3 0 abcd
beq 2 3 5
ori 1 0 d   #a0h
ori 1 0 e
ori 1 0 f
ori 1 0 10
ori 1 0 11 #b0h
ori 1 0 12
ori 3 0 abcb
beq 2 3 fffb

# bne - branch on not equal
nop
nop
addi 3 3 1
nop
bne 2 3 fffd
nop       # run for 3300

# blez - branch on less than or
equal to zero
addi 2 0 fffe
addi 2 2 1
blez 2 fffe
nop

# bgtz - branch on greater than
zero
ori 2 0 3
addi 2 2 ffff
bgtz 2 fffe
nop

# bltz - branch on less than zero
addi 2 0 fffd
addi 2 2 1
bltz 2 fffe
nop       # run for 5000

# bgez - branch on greater than
or equal to zero
ori 2 0 2
addi 2 2 ffff
bgez 2 fffe
nop

# bltzal - branch on less than
zero and link
addi 2 0 fffd
addi 2 2 1
bltzal 2 fffe
nop      # run for 6200

# bgezal - branch on greater than
or equal to zero and link
ori 2 0 2
addi 2 2 ffff
bgezal 2 fffe
nop      # run for 6800

halt
```

LS.TEST - LOAD AND STORE

```
# test file for load and store
instructions

lui 1 abcd
lui 2 5678
ori 1 1 ef01
ori 2 e47f
sw 1 4000 0
sh 1 4004 0
sb 1 4006 0
sb 2 4007 0
sb 0 4008 0
swl 2 4009 0
swr 2 400c 0
sb 0 400d 0
sh 0 400e 0
sh 0 4010 0
swl 2 4012 0
swr 2 4015 0
swl 2 401b 0
swr 2 401e 0
swl 2 4020 0
swr 2 4027 0
#
lw 3 4000 0
nop
lw 3 4004 0
nop
lw 3 4008 0
nop
lw 3 400c 0
nop
lw 3 4010 0
nop
lw 3 4014 0
nop
lw 3 4018 0
nop
lw 3 401c 0
nop
lw 3 4020 0
nop
lw 3 4024 0
nop
#
lb 3 4000 0
nop
lb 3 4001 0
nop
lb 3 4002 0
nop
lb 3 4003 0
nop
lbu 3 4000 0
nop
lbu 3 4001 0
nop
lbu 3 4002 0
nop
lbu 3 4003 0
nop
lh 3 4004 0
nop
lh 3 4006 0
nop
lhu 3 4004 0
nop
lhu 3 4006 0
nop
#
ori 3 0 0
lwl 3 4021 0
nop
lwr 3 4024 0
nop
#
ori 3 0 0
lwl 3 4022 0
nop
lwr 3 4025 0
nop
#
ori 3 0 0
lwl 3 4023 0
nop
lwr 3 4026 0
nop
#
lwl 3 4020 0
nop
lwr 3 4023 0
nop
halt
```

MD.TEST - MULTIPLICATION AND DIVISION

```
# test file for multiply and
divide instructions

# mult - multiply
ori 1 0 8
ori 2 0 9
mult 1 2    # 8*9
mult 2 1

addi 2 0 fff7
mult 1 2    # 8*-9
mult 2 1

addi 1 0 fff8
mult 1 2    # -8*-9
mult 2 1

ori 2 0 9
mult 1 2    # -8*9
mult 2 1

ori 1 0 0
ori 2 0 0
mult 1 2    # 0*0

ori 1 0 a
mult 1 2    # 0*a

lui 1 7fff
ori 1 1 ffff
ori 2 0 0
mult 1 2    # 7fff_ffff * 0
```

```
ori 2 0 1
mult 1 2   # 7fff_ffff * 1

ori 2 0 10
mult 1 2   # 7fff_ffff * 10h

lui 1 8000
ori 2 0 0
mult 1 2   # 8000_0000 * 0

ori 2 0 1
mult 1 2   # 8000_0000 * 1

ori 2 0 10
mult 1 2   # 8000_0000 * 10h

addi 1 0 ffff
ori 2 0 0
mult 1 2   # ffff_ffff * 0

ori 2 0 1
mult 1 2   # ffff_ffff * 1

ori 2 0 10
mult 1 2   # ffff_ffff * 10h

addi 2 0 ffff
mult 1 2   # ffff_ffff *
ffff_ffff

# multu - multiply unsigned
ori 1 0 8
ori 2 0 9
multu 1 2   # 8*9
multu 2 1

addi 2 0 fff7
multu 1 2   # 8*-9
multu 2 1

addi 1 0 fff8
multu 1 2   # -8*-9
multu 2 1

ori 2 0 9
multu 1 2   # -8*9
multu 2 1

ori 1 0 0
ori 2 0 0
multu 1 2   # 0*0

ori 1 0 a
multu 1 2   # 0*a

addi 1 0 ffff
ori 2 0 0
multu 1 2   # ffff_ffff * 0

ori 2 0 1
multu 1 2   # ffff_ffff * 1

ori 2 0 10
multu 1 2   # ffff_ffff * 16
(10h)

ori 2 0 40
multu 1 2   # ffff_ffff * 64
(40h)

ori 2 0 100
multu 1 2   # ffff_ffff * 256
(100h)

addi 2 0 ffff
multu 1 2   # ffff_ffff *
ffff_ffff


# div - divide
ori 1 0 48
ori 2 0 9
div 1 2     # 72d/9d

addi 2 0 fff7
div 1 2     # 72d/-9d

addi 1 0 ffb8
div 1 2     # -72d/-9d

ori 2 0 9
div 1 2     # -72d/9d

#
ori 1 0 49
ori 2 0 9
div 1 2

addi 2 0 fff7
div 1 2

addi 1 0 ffb7
div 1 2

ori 2 0 9
div 1 2

# divu - divide unsigned
ori 1 0 48
ori 2 0 9
divu 1 2     # 72d/9d

addi 2 0 fff7
divu 1 2     # 72d/-9d

addi 1 0 ffb8
divu 1 2     # -72d/-9d

ori 2 0 9
divu 1 2     # -72d/9d
#
ori 1 0 4b
ori 2 0 9
divu 1 2

addi 2 0 fff7
divu 1 2

addi 1 0 ffb5
divu 1 2

ori 2 0 9
divu 1 2  # run for 6000

lui 1 acef
lui 2 48a2
ori 1 1 52ac
ori 2 2 9165
divu 1 2   # run for 7200

# mfhi - move from hi
mfhi 3

# mthi - move to hi
ori 3 0 abcd
mthi 3

# mflo - move from lo
mflo 3

# mtlo - move to lo
ori 3 0 1234
mtlo 3

halt
```

MV.TEST - MOVE TO/FROM HI/LO REGISTERS

```
lui 3 abcd
ori 3 3 1234

# mthi - move to hi
mthi 3

# mfhi - move from hi
mfhi 4

# mtlo - move to lo
mtlo 3

# mflo - move from lo
mflo 5

halt
```

S.TEST - SHIFT

```
# test file for shift functions

# sll - shift left logical
sll 0 0 0
sll 0 0 4
sll 1 0 0
sll 1 0 4
sll 0 1 0
sll 0 1 4
sll 2 1 0
sll 2 1 4
ori 1 0 1
sll 2 1 1
sll 2 1 2
sll 2 1 3
sll 2 1 4
sll 2 1 8
sll 2 1 16
sll 2 1 30
sll 2 1 31    # run for 1300

# srl - shift right logical
srl 1 2 1
srl 1 2 2
srl 1 2 3
srl 1 2 4
srl 1 2 8
srl 1 2 16
srl 1 2 30
srl 1 2 31

# sra - shift right arithmetic
sra 0 0 0
sra 0 0 8
lui 1 8000
ori 1 1 1
sra 2 1 1
lui 1 2222
ori 1 1 2222
sra 2 1 1
lui 1 4444
ori 1 1 4444
sra 2 1 2
lui 1 8888
ori 1 1 8888
sra 2 1 1
sra 2 1 2
sra 2 1 4    # run for 2800

# sllv - shift left logical
variable
ori 2 0 0
ori 1 0 f
ori 3 0 4
sllv 2 1 3
ori 3 0 10
sllv 2 1 3

# srlv - shift right logical
variable
ori 2 0 0
lui 1 f000
ori 1 1 f
ori 3 0 2
srlv 2 1 3

# srav - shift right arithmetic
variable
ori 2 0 0
lui 1 8888
ori 1 1 8888
ori 3 0 4
srav 2 1 3
halt
```

NOTE: ONLY BEHAVIORAL MODEL TEST PROGRAMS USE THE *HALT* INSTRUCTION. *HALT* IS REPLACED WITH THE *BREAK* INSTRUCTION FOR DATAFLOW AND STRUCTURAL MODELS.